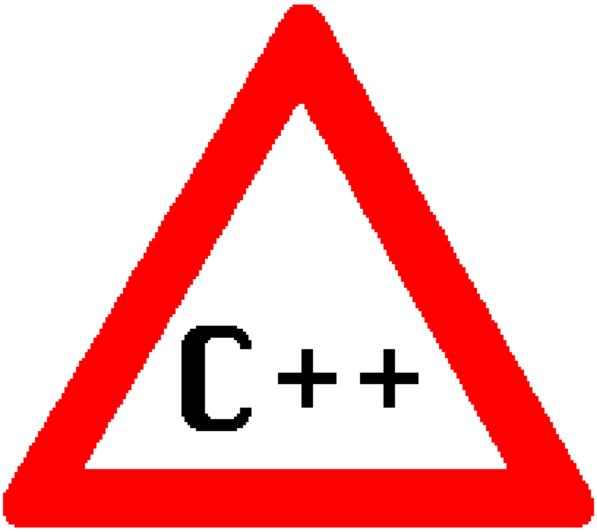
Universität Hildesheim

Skripten des Fachbereichs
Mathematik, Informatik, Naturwissenschaften
Nr. ***



Michael Uelschen

Die C++ Fibel

1.Auflage

Hildesheim · Januar 1996
Institut für Betriebssysteme und Rechnerverbund

Anschrift des Autors:
Dipl.–Inform. Michael Uelschen
Institut für Thermische Energietechnik
Fachgebiet Strömungsmaschinen
Fachbereich Maschinenbau
Universität Gesamthochschule Kassel
D–34109 Kassel

email: `uelschen@hrz.uni-kassel.de`

Bibliographische Angaben:
Uelschen, Michael:
Die C++ Fibel /Michael Uelschen
1.Auflage – Hildesheim: Universität Hildesheim, 1996
(Skripten des Fachbereichs Mathematik, Informatik, Naturwissenschaften Nr. ***)

ISSN ****–****

Satz: Michael Uelschen

Ein LaTeX2e Dokument.

Informationen zu den Skripten des Fachbereichs Mathematik, Informatik, Naturwissenschaften:
Prof. Dr. Günther Stiege
Institut für Betriebssysteme und Rechnerverbund
Universität Hildesheim
Samelsonplatz 1
D–31141 Hildesheim

email: `stiege@informatik.uni-hildesheim.de`

Universitätsbibliothek
Universität Hildesheim
Postfach 101363
31113 Hildesheim

Vorwort

Die Verwendung objektorientierter Konzepte bei der Erstellung von Software gewinnt stetig an Bedeutung. Vermehrt wird in der Praxis Kenntnisse in objektorientierter Programmierung und der Sprache C++ von Diplom–Informatikerinnen und Diplom–Informatikern verlangt. In der Ausbildung an der Universität Hildesheim wird dieses u.a. durch ein Programmierpraktikum im Grundstudium vermittelt.

Die Programmiersprache C++ verbindet objektorientierte Konzepte und maschinennahe Programmierung. Bjarne Stroustrup schuf an den AT&T Labs in den achtziger Jahren C++ als Erweiterung der Sprache C.

Im April 1995 ist der Entwurf zum ISO C++ Standard veröffentlicht worden. Noch 1996 oder spätestens 1997 ist mit einem endgültigen Standard zu rechnen. Allerdings bieten die bisherigen (kommerziellen) C++ Systeme noch keine vollständige Implementierung des geforderten Standards. In dieser Fibel wird an einigen Stellen durch Randnotizen auf entsprechende Kapitel und Abschnitte des ISO Dokumentes verwiesen.

Die meisten Bücher, die sich mit der Programmiersprache C++ befassen, beschränken sich aut die Sprache als solche. In diesem Skript ist der Versuch unternommen worden, auch auf das Umfeld, welches in einem UNIX™ System zu finden ist, einzugehen. Hierzu gehört u.a. die Benutzung des `emacs` *Editors und die Erstellung sogenannter* `Makefile` *Dateien. Ein weiterer Abschnitt befaßt sich mit der Fehlersuche und dem Einsatz von Testhilfen (Debugger).*

Viele Aspekte konnten nur ansatzweise vorgestellt werden. Diese Unterlagen können kein Lehrbuch ersetzen. Zur weiteren Vertiefung sei daher auf die reichlich vorhandene Literatur verwiesen.

Ich danke allen, die mir bei der Erstellung dieser Unterlagen behilflich waren. Insbesondere gilt mein Dank Prof. Dr. Stiege, der mir die Veröffentlichung im Rahmen der Skripte der „Roten Reihe" ermöglicht hat.

Im Sommersemester 1995 wurde ein Grundpraktikum C und C++ für das vierte Semester im Studiengang Informatik erstmalig durchgeführt.

Hildesheim, im Januar 1996 — Michael Uelschen

Inhaltsverzeichnis

Abbildungsverzeichnis

Lektion 1

Vorbemerkungen

In der ersten Lektion wird ein Überblick über Inhalt und Ziel der vorliegenden **C++ Fibel** gegeben.

1.1 Allgemeines

Diese Fibel enthält eine Sammlung von Aufgaben, die unter Benutzung weiterer Unterlagen selbständig zu lösen sind. Die Fibel ersetzt kein Lehrbuch, sondern soll dazu dienen, die **objektorientierte Programmiersprache** (OOPS) **C++** anhand von praktischen Aufgaben zu erlernen. Des weiteren werden verschiedene **Werkzeuge zur Programmentwicklung** beschrieben.

Objektorientiert ist ein inzwischen weit verbreitetes Schlagwort im Bereich der Informationstechnologie.
Oftmals wird der Eindruck vermittelt, daß objektorientierte Techniken etwas grundsätzlich Neues sind bzw. nur durch Anwendung dieser Probleme gelöst werden können.
Das in C++ unterstützte Klassenkonzept wurde bereits in **Simula67** verwendet. Die Möglichkeit, Operatoren und Funktionen zu überladen, ist Bestandteil von **Algol68**. Auch lassen sich mit objektorientierten Programmiersprachen keine anderen Probleme lösen als mit herkömmlichen, z.B. mit prozeduralen Sprachen wie **Pascal** oder **Modula–2**. Vielmehr erheben OOPS den Anspruch, Software, insbesondere größere Projekte, „effizienter" zu erstellen. In den weiteren Lektionen wird dieses konkretisiert.
Die Erstellung von Software gliedert sich im wesentlichen in drei Abschnitte:

Analyse Bestimmung und Untersuchung des zu lösenden Problemkreises

Design Konstruktion des Systems in Hinblick auf eine Implementierung

Implementierung Schreiben und Testen des Programms

C++ ist eine Erweiterung der Programmiersprache **C** (++ ist der Inkrementierungsoperator in C) und enthält im wesentlichen dieselben Konstrukte (eine 100–prozentige Obermenge ist C++ nicht, das sollte aber hier nicht weiter stören). Folglich läßt sich in C++ auch nicht objektorientiert programmieren. Die OO-Komponente entsteht also nicht automatisch, sondern ist nur durch eine **objektorientierte Analyse** (OOA) und **Design** (OOD) möglich.
Deshalb soll in dieser Fibel insbesondere auch auf solche Analyse und Entwurfstechniken eingegangen werden. Hierbei wird im wesentlichen ein pragmatischer Ansatz gewählt; Formalismen sind Gegenstand weiterer Lehrveranstaltungen (z.B. Informationssysteme).

Die Abbildung 1.1 aus [Str94a, Seite 6] zeigt einen Ausschnitt aus der Evolution einiger

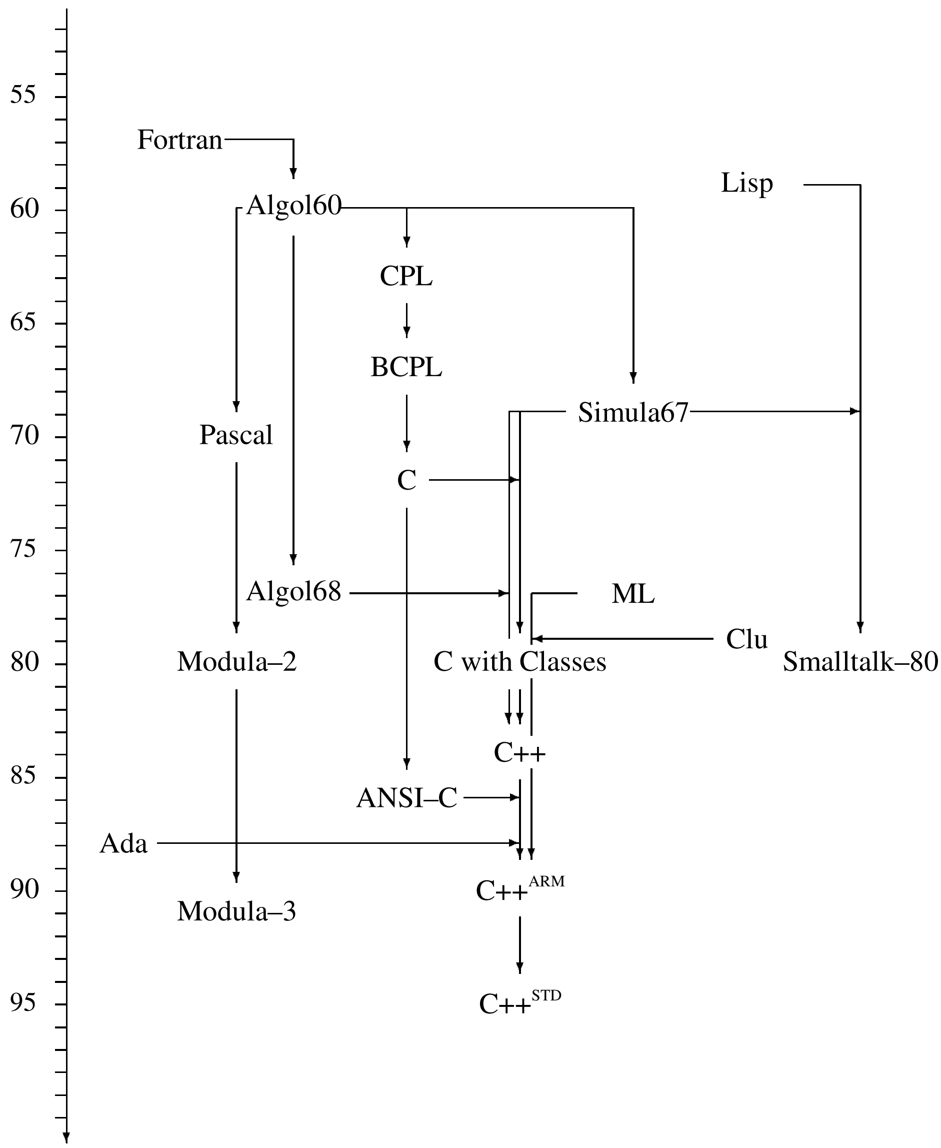


Abbildung 1.1: Die Evolution von C und C++

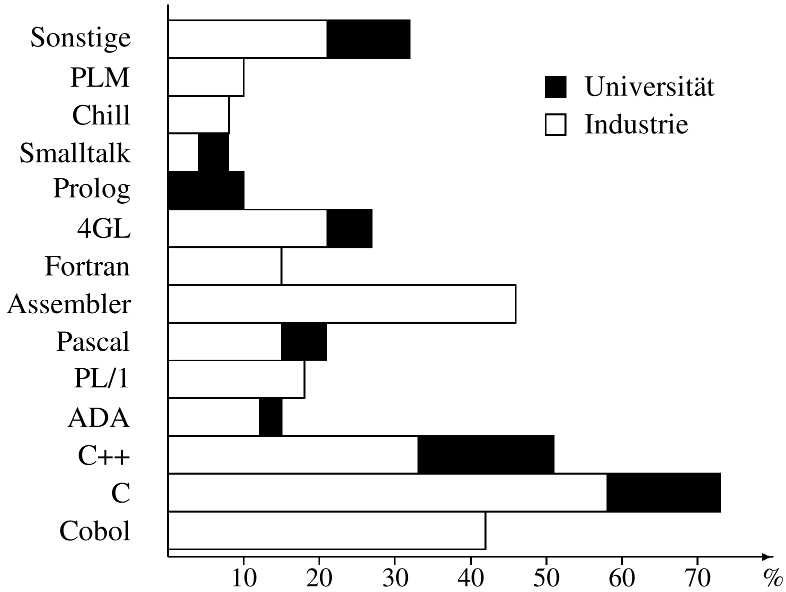


Abbildung 1.2: Programmiersprachen in Universität und Industrie

gebräuchlicher Programmiersprachen, insbesondere die Entwicklung von C und C++. Die Sprachen sind zum Zeitpunkt der ersten brauchbaren Implementierung eingeordnet; ein Algol68–Übersetzer war erstmals im Jahr 1977 verfügbar.

Die Programmiersprache **C** wurde Anfang der 70'er Jahre von **Dennis M. Ritchie** entworfen und für das Betriebssystem **UNIX**™ implementiert.
Vorläufer waren die Sprache **BCPL** von **Martin Richards** und die Sprache **B** von **Ken Thompson** (vgl. Abbildung 1.1 ohne B).
Die in [KR83] beschriebene Version wird auch oftmals als **K&R C** bezeichnet, nachdem vom ANSI–Komitee X3J11 eine Sprachbeschreibung **ANSI–C** als Standard veröffentlicht wurde.

Bjarne Stroustrup beschreibt in [Str94a] ausführlich die Entwicklung der Sprache C++. Die Einflüsse sind vielfältig; hauptsächlich wurde sich anfangs an den bereits existierenden Sprachen C und Simula67 orientiert.
1990 erschien das Buch [ES90] von Ellis und Stroustrup, welches eine Art Quasi–Standard **C++**$^{\text{ARM}}$ beschreibt. Dieser dient als Grundlage des ANSI–Komitee X3J16, welches sich mit der Standardisierung von C++ beschäftigt. Es ist geplant, 1996 den Standard **C++**$^{\text{STD}}$ zu veröffentlichen.

In der C++ Schmiede der **Bell Labs** von **AT&T** ist inzwischen auch eine zweite objektorientierte Sprache basierend auf C entwickelt worden. Sie hat den etwas eigenwilligen Namen **C+@**[1].Eine Einführung ist in [Wag95] zu finden.

Neben C++ existieren eine Reihe von weiteren OOPS, wie z.B. **Smalltalk** und **Eiffel**, die jedoch (noch) ein Nischendasein fristen. C++ verbindet die Vorteile von C, die Möglichkeit einer maschinennahen Programmierung und die

[1] Ausgesprochen wird dieses als *cat*.

Ideen der objektorientierten Gedankenwelt.
Neuerliche Popularität erlangen objektorientierte Betrachtungsweisen und Programmiersprachen mit der zunehmenden Verbreitung von fensterorientierten Bedienoberflächen, da sich hier OO-Konzepte offensichtlich anbieten.

C++ ist nicht nur im akademischen Bereich zu finden, sondern wird vermehrt in der Industrie und Wirtschaft verlangt und eingesetzt (vgl. Abbildung 1.2 aus [SL94a][2]).
Die Beschäftigung mit C++ hat also ein begründetes Interesse. Diese Fibel soll dazu beitragen, C++ in den wesentlichen Konzepten zu verstehen und anzuwenden.

1.2 Struktur

Die Fibel gliedert sich in 12 Lektionen. Eine sequentielle und umfaßende Vorgehensweise ist nicht notwendig. Sie richtet sich im wesentlichen nach den verfügbaren Vorkenntnissen und an der Zielvorstellung.
Im folgenden wird der Inhalt der einzelnen Lektionen kurz umrissen.

Lektion 1 Vorbemerkungen In dieser Lektion befinden sich einführende Informationen zu C++ und verschiedene Möglichkeiten der Hilfestellung.

Lektion 2 Umgebung Um praktisch mit C++ zu arbeiten, ist eine Programmierumgebung zum Editieren, Übersetzen und Binden notwendig. In der zweiten Lektion werden dazu ein paar Hinweise gegeben.

Lektion 3 Fehlersuche und Testhilfen In dieser Lektion wird gezeigt, wie Fehler entdeckt und dann behoben werden können. Insbesondere wird auf verschiedene Debugger eingegangen.

[2] Die angegebenen Prozentzahlen beziehen sich stets auf die Gesamtheit der Befragten. Die Aussage ist nicht unbedingt repräsentativ, sondern soll Tendenz und Überblick aufzeigen.

Lektion 4 Analyse und Design Analyse und Design sind die wesentlichen Schritte in der Entwicklung von Software. In der vierten Lektion wird eine kurze Einführung in die objektorientierte Gedankenwelt gegeben.

Lektion 5 Einführung Die weiteren Lektionen beschäftigen sich mit der Implementierung, also der dritten Phase in der Entwicklung. In der fünften Lektion werden die grundlegenden Konzepte von C++ vorgestellt.

Lektion 6 Konstruktoren und Destruktoren In dieser Lektion wird auf besondere Funktionen zur Konstruktion und zur Zerstörung von Objekten eingegangen.

Lektion 7 Vererbung Ein grundlegendes Konzept in der objektorientierten Welt bildet die Vererbung, auf die in dieser Lektion eingegangen wird.

Lektion 8 Virtuelle Methoden und abstrakte Basisklassen Durch die Einführung von virtuellen Methoden ist es möglich, Konzepte transparenter darzustellen.

Lektion 9 Schablonen Eine Erweiterung der bisherigen Konzepte bieten generische oder parametrisierte Typen und Funktionen, die als Schablonen (Templates) bezeichnet werden.

Lektion 10 Fehlerbehandlung In dieser Lektion wird eine nicht objektorientierte Erweiterung in C++ besprochen, die es erlaubt, Fehlersituationen komfortabel zu handhaben.

Lektion 11 Projekt In der vorletzten Lektion wird ein etwas größeres Software–Projekt vorgestellt.

Lektion 12 Vergleich von C++ und C Im Gegensatz zu anderer Literatur wird von C++

ausgehend auf die Unterschiede zu C eingegangen.

Fernerhin befindet sich am Ende ein Literaturverzeichnis. In den Anhängen sind weiterführende Informationen zu C und C++ aufgeführt.

1.3 Hilfe

In diesem Abschnitt werden verschiedene Möglichkeiten vorgestellt, um bei auftretenden Problemen Hilfestellungen zu erhalten.

Online–Handbuch Für viele Kommandos, Programme und auch C– bzw. C++ –Bibliotheken ist ein Online–Handbuch verfügbar. Dieses wird mit dem Befehl `man` und dem gesuchten Begriff als Parameter aufgerufen. Mit `man man` erhält man also Hilfe über das Online–Handbuch. Der Befehl `apropos` bietet die Möglichkeit, zu einem Stichwort die Handbuchseiten aufzulisten, die Informationen zu diesem haben. Ein weiteres Online Hilfesystem steht unter dem Namen `info` zur Verfügung.

Literatur Zur Programmiersprache C/C++ sind hinreichend viele Bücher und Artikel verfaßt worden. Eine kleine Auswahl sind z.B. [Str91], [Cop92], [Lip91] und [BT89]. Fernerhin sind zu den (kommerziellen) Produkten ausführliche Handbücher vorhanden.
Die Zeitschriften **The C++ Report** und **C/C++ Users Journal** widmen sich ausführlich der Programmierung mit C und C++. Das **Journal of Object Oriented Programmierung** bietet Informationen zur objektorientierten Programmierung und zu verschiedenen Programmiersprachen.

Newsgroup Es existieren verschiedene Newsgroups im USENET, die sich mit der Programmierung allgemein und mit C bzw. C++ im besonderen befassen. Hier seien u.a. `comp.lang.c++` bzw. die deutsche Variante `de.comp.lang.c++` genannt.
Der Standardisierungsprozeß wird in `comp.std.c++` diskutiert. Die Qualität der Beiträge dieser Gruppe zeichnet sich u.a. dadurch aus, daß viele Mitglieder aus dem Standardisierungskomitee und dem Umfeld, wie z.B. Bjarne Stroustrup, sich aktiv beteiligen.

FAQ Aus den Newsgroups sind auch die sogenannten FAQ (Frequently Asked Questions) hervorgegangen, die die am häufigsten gestellten Fragen und deren Antworten zusammenfassen.

WWW Das verteilte Informationssystem World Wide Web (WWW) bietet eine Reihe von Informationen, z.B. Online–Programmierkurse, zu der Programmiersprache C/C++ und dem zugehörigen Umfeld. Im Anhang A.2 sind einige Verweise aufgeführt.

EMAIL Fragen und Anmerkungen zu dieser Fibel können per email an den Autor

`uelschen@hrz.uni-kassel.de`

gerichtet werden.

1.4 Voraussetzungen

Die Programmiersprache C war anfangs nur auf Rechnern mit dem UNIX™–Betriebssystem verfügbar. Inzwischen ist diese jedoch auf sämtlichen Rechnerplattformen, angefangen bei Kleinstrechnern bis hin zu Supercomputern, zu finden. Ebenfalls ist C++ auf die meisten Plattformen portiert worden und sollte somit den meisten Anwenderinnen und Anwendern zur Verfügung stehen.

In dieser Fibel wird implizit von einer UNIX™–Umgebung ausgegangen, insbesondere betrifft dieses die vorgestellten Werkzeuge.
Es sollte jedoch ohne weiteres möglich sein, die meisten Aufgaben auch unter MS-DOS™ oder OS/2™ nachzuvollziehen. Die dort verfügbaren Programmiersysteme haben oftmals den Vorteil einer integrierten Entwicklungsumgebung, die in einer universitären UNIX™ Umgebung nicht vorhanden ist[3]. Schwierigkeiten bezüglich der Portabilität zwischen verschiedenen Systemen sind höchstwahrscheinlich nur bei der Benutzung von Schablonen (templates) und der Fehlerbehandlung (exception handling) zu erwarten, da der in [ES90] beschriebene Standard noch nicht von allen Herstellern implementiert wurde.

Von den Anwenderinnen und Anwendern wird vorausgesetzt, daß diese mindestens eine (imperative) Programmiersprache beherrschen und die grundlegenden Algorithmen und Datenstrukturen, wie z.B. Suchen in Listen, Durchlauf in binären Bäumen, verstanden haben.
Studierende im vierten Semester der Informatik an der Universität Hildesheim sollten daher diese Anforderungen erfüllen.

1.5 Was (noch) fehlt

Viele Eigenschaften von C/C++ und weitere UNIX™ Werkzeuge sind in diesem Skript nur unvollständig bzw. überhaupt nicht erwähnt worden. Hierzu gehören u.a.:

`dbmalloc` Sogenannte Speicherfehler, z.B. das mehrfache Freigeben von allokierten Speicherbereichen oder der Zugriff auf bereits freigegeben Speicher, sind schwer zu beheben. Die `dbmalloc` Bibliothek implementiert die Befehle zur Allokation und Freigabe neu. Zur Laufzeit wird der Speicherzugriff überwacht und es werden Unregelmäßigkeiten aufgedeckt.

`lex` und `yacc` Diese beiden Werkzeuge sind bei der Implementierung von Kompilern nützlich.

Systembefehle Der Aufruf von Systembefehlen, z.B. zur Interprozeßkommunikation, ermöglicht es, die Funktionalität eines UNIX™ Systems auszunutzen.

...

[3] Gilt für die Ausstattung an der Universität Hildesheim.

Lektion 2

Umgebung

In den folgenden Abschnitten werden einige Werkzeuge, die zur Programmerstellung benötigt werden, dargestellt. Die hier aufgeführten Funktionen der verschiedenen Programme erheben keinen Anspruch auf Vollständigkeit, sondern sollen einen (schnellen) Einstieg in die Programmierung erlauben.

2.1 Programmerstellung

Den Bereich der reinen Programmerstellung umfaßt die Eingabe des Quelltextes, das Übersetzen und das Binden.
Abbildung 2.5 zeigt den Ablauf der Programmerstellung (ohne Editor).
In den folgenden Abschnitten wird der Präprozessor `cpp`, der Übersetzer `CC` (bzw. `g++`, `cc` und `gcc`) und der Binder `ld` vorgestellt. Auf eine Beschreibung des Assemblers `as` wird an dieser Stelle verzichtet.
Im einfachsten Fall werden Quelltextdateien (mit der Endung `.h` und `.cc`) eingelesen und, sofern keine Fehler auftreten, ein ausführbares Programm (standardmäßig `a.out`) erzeugt.
Jedoch ist es möglich, eine Reihe von teilbearbeiteten Dateien zu erzeugen, die an anderer Stelle weiterverarbeitet werden können.
Etwas außerhalb der gewöhnlichen Verarbeitung steht das `make`–Werkzeug, welches es erlaubt, gewisse Abläufe der Programmerstellung zu vereinfachen.
Zuerst werden jedoch einige Möglichkeiten zur Eingabe des Quelltextes vorgestellt.

2.2 Texteditoren

Nachfolgend werden die beiden Editoren `vi` und `emacs` beschrieben. Diese zeichnen sich durch ihren großen Befehlsumfang aus; haben aber den Nachteil einer etwas gewöhnungsbedürftigen Bedienung.
Dieses gilt besonders dann, wenn die Programmiererin bzw. der Programmierer an eine fensterorientierte Bedienoberfläche gewöhnt ist. Fenster, Menübalken, Knöpfe usw. bietet `vi` (der `emacs` nur eingeschränkt in der X11 Version) **nicht**.
Die Beherrschung eines dieser Werkzeuge ermöglicht jedoch eine schnelle Eingabe und Bearbeitung von Daten bzw. Quelltexten. Es zeigt sich, daß die Eingabe einer kurzen Zeichenfolge per Tastatur zur Befehlsausführung wesentlich schneller ist als die Bedienung mit der Maus. Oftmals muß man sich nämlich hier durch eine Vielzahl von Menüs und Unterpunkten *hindurchklicken*.

In den nachfolgenden Abschnitten werden zur Verdeutlichung einige Tastatursymbole verwendet, die vorher kurz erläutert werden sollen. Einzelne Tasten sind durch einen Kasten symbolisiert, z.B. $\boxed{\texttt{a}}$ bedeutet, daß die Taste

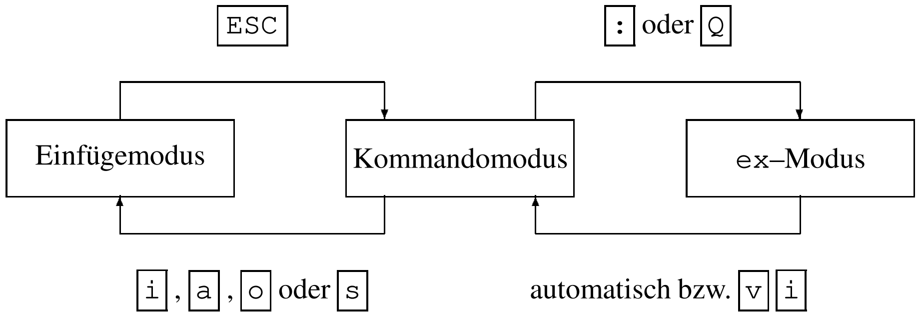


Abbildung 2.1: Die einzelnen Modi des `vi`

mit einen `a`[1] gedrückt werden muß, um eine im Text beschriebene Wirkung zu erzielen.
Um Großbuchstaben zu erhalten, muß während des Tastendruckes die Umschalt– (shift, caps) Taste gedrückt werden. So bedeutet also zum Beispiel `A` , daß zwei Tasten (Umschalttaste und `a`) betätigt werden müssen.
Einige besondere Tasten und Tastenfolgen werden im weiteren Text wie folgt graphisch dargestellt:

` `	Leertaste (space,blank)
`ESC`	Escape–Taste
`CTRL`	Steuerungstaste (strg,ctl,control)
`F 8`	Funktionstaste 8 (F8)
`RETURN`	Return–Taste (enter)
`DEL`	Löschen (Entfernen) –Taste
`TAB`	Tabulatortaste
`CTRL` + `a`	Steuerungstaste und `a` **gleichzeitig** drücken.
`v` `i`	die Tasten `v` und `i` **nacheinander** drücken.

[1] Die Tastaturen sind in der Regel mit Großbuchstaben beschriftet, wenn auch eigentlich Kleinbuchstaben bei einer Betätigung erscheinen.

2.2.1 Der Editor `vi`

Im folgenden wird eine kurze Einführung in den *Standard*–Editor `vi` („visual“) gegeben, der wohl in jedem UNIX™ ähnlichem System zu finden ist. Eine ausführlichere Beschreibung ist z.B. in dem Buch [Gul88] oder auch in dem Skript [Hil91] zu finden.

2.2.1.1 Allgemeines

Beim Aufruf `vi` können u.a. folgende Parameter optional angegeben werden:

–R `vi` lädt Dateien nur zum Lesen (read–only)

+command führt das angegebene Kommando *command* aus.

So startet `vi Makefile` den Editor und liest zu Beginn die Datei `Makefile` ein. `vi +/OBJECTS Makefile` plaziert den Cursor auf den Anfang der ersten Zeile im `Makefile`, die die Zeichenkette `OBJECTS` enthält (das Suchkommando wird weiter hinten beschrieben).

Auf eine bildliche Darstellung des `vi` wurde

verzichtet; nach dessen Aufruf erscheint nur eine Statusanzeige in der untersten Zeile auf dem Bildschirm bzw. in dem entsprechenden Fenster. Tildesymbole ~ in der ganz linken Spalte einer jeden Zeile zeigen gegebenenfalls das Ende des Textes an.
Der `vi` kennt drei verschiedene Modi:

Einfügemodus erlaubt die Eingabe des eigentlichen Textes.

Kommandomodus erlaubt die Ausführung einer Vielzahl von Befehlen, z.B. Suchen und Ersetzen.

`ex`–Modus ermöglicht die Ausführung von Befehlen des `ex`-Editors.

Der letzte Punkt bedarf einer weiteren Erklärung. `vi` und `ex` sind zwei Editoren, die miteinander kooperieren. So hat der `vi` zum Beispiel keine Befehle zur Dateibearbeitung, hierzu werden die entsprechenden Befehle des `ex`–Editors aufgerufen. `ex` selber ist ein Zeileneditor[2], auf den hier (soweit es den `vi` nicht betrifft) nicht weiter eingegangen werden soll.
Einfügemodus und Kommandomodus werden oftmals zusammen als **`vi`–Modus** bezeichnet.

Im Kommandomodus bestehen die Befehle im allgemeinem aus einem einzelnen Tastendruck bzw. aus einer kurzen Folge von Tastendrücken. Die Ausführung des Befehls erfolgt, sobald eine hinreichend gültige Tastenkombination eingegeben worden ist. Eine explizite Bestätigung ist nicht nötig und möglich. Erkennt der `vi` eine fehlerhafte Eingabe, so ertönt ein Warnton.
Hingegen werden alle Befehle im `ex`–Modus durch einen Abschluß mit der `RETURN` –Taste bestätigt und in der Statuszeile angezeigt. Ebenso müssen die beiden Befehle `/` und `?` durch ein `RETURN` abgeschlossen werden.
Befehle des `ex`–Editors müssen mit einem **:** eingeleitet werden. Hierzu wird im Kommandomodus die `:` –Taste betätigt. Nach der Eingabe und dem Abschluß durch die `RETURN` –Taste wird automatisch in den Kommandomodus zurück gewechselt.
Um dauerhaft Befehle im `ex`–Modus einzugeben, muß mit dem `Q` Befehl vom Kommandomodus in diesen gewechselt worden sein. Ein Rücksprung ist dann durch `v` `i` möglich.

Die Abbildung 2.1 stellt den Zusammenhang zwischen den einzelnen Modi dar[3]. Der Umstand, daß Einfüge– und Kommandomodus optisch nicht zu unterscheiden sind, führt oftmals zu Beginn zu Schwierigkeiten[4]: eine fälschlicherweise im Kommandomodus als Text gedachte Zeichenfolge wird als Kommandosequenz ausgeführt, woraufhin u.U. eine nicht beabsichtigte Veränderung eintritt. Entsprechendes gilt, wenn im Einfügemodus eine Befehlssequenz abgesetzt wird.

Eine Reihe von Kommandos können auf sogenannte **Objekte** angewendet werden.

Tastenfolge	Bedeutung
` `	Einzelnes Zeichen
`w`	Wort ohne Sonderzeichen
`W`	Wort mit Sonderzeichen
`b`	Vorheriges Wort ohne Sonderzeichen
`B`	Vorheriges Wort mit Sonderzeichen
`^`	Bis zum Anfang der Zeile
`$`	Bis zum Ende der Zeile

[2] Im Prinzip ist die Funktionalität des `ex` eine Obermenge der des `ed`–Zeileneditors.

[3] Der Übergang vom Kommando– in den Einfügemodus ist durch eine Reihe von Befehlen möglich. Aus Übersichtgründen sind in der Abbildung nur einige aufgeführt.

[4] Durch Anpassung einer internen Option ist es doch möglich.

(	Bis zum Anfang des Satzes
)	Bis zum Ende des Satzes
{	Bis zum Anfang des Absatzes
}	Bis zum Ende des Absatzes
[[	Bis zum Anfang des Abschnitts
]]	Bis zum Ende des Abschnitts

Die einzelnen Objekte sind definiert als:

Wort Folge von Buchstaben und Ziffern ohne Leerzeichen, Tabulatoren oder Neue–Zeile Zeichen.

Zeile Folge von Zeichen, die mit einem Neue–Zeile Zeichen abgeschlossen wird.

Satz Folge von Worten, die mit `.` (Punkt), `,` (Komma), `!` (Ausrufungszeichen) oder `?` (Fragezeichen) **und** zwei Leerzeichen oder einem Neue–Zeile Zeichen abgeschlossen wird.

Absatz Folge von Worten, die durch eine Leerzeile begrenzt wird.

Abschnitt Folge von Worten, die durch ein Neue–Seite Zeichen als erstes Zeichen in einer Zeile begrenzt wird.

Ist eine Zeile länger als die darstellende Breite des Bildschirms bzw. des Fensters, so wird die Zeile in der Darstellung umgebrochen, d.h. Bildschirmzeilen sind nicht unbedingt gleich Textzeilen. Dieses ist besonders in Hinblick auf die Bewegung des Cursors wichtig, da der Cursor **immer** bezüglich der Textzeilen bewegt wird.

Cursorbewegung Es gibt eine Reihe von Möglichkeiten, den **Cursor** (Arbeitszeiger) im Text zu bewegen. Hierzu muß gegebenfalls zuerst in den Kommandomodus umgeschaltet werden.

Als erstes kann der Cursor durch die Eingabe der entsprechenden Taste um eines der aufgeführten Objekte bewegt werden. Also durch W wird zum Beispiel auf das erste Zeichen des nächsten Wortes gesprungen.
Durch die optionale Angabe einer *Anzahl* wird dieses dementsprechend häufig wiederholt, 6 B bewegt den Cursor 6 Wörter zurück.
^ springt an den Anfang, $ an das Ende der aktuellen Zeile.

In der nachfolgenden Tabelle sind weitere Positionierungskommandos[5] aufgeführt.

Tastenfolge	Auswirkung
Anzahl ␣	bewegt den Cursor *Anzahl* Zeichen nach **rechts**.
Anzahl h	bewegt den Cursor *Anzahl* Zeichen nach **links**.
Anzahl k	bewegt den Cursor *Anzahl* Zeichen nach **oben**.
Anzahl j	bewegt den Cursor *Anzahl* Zeichen nach **unten**.
0	bewegt den Cursor zum **ersten sichtbaren Zeichen** in der Zeile.
nr G	springt mit dem Cursor in die Zeile *nr*. Ist keine Zeilennummer angegeben, so wird der Cursor ans Ende des Textes plaziert.

Des weiteren gibt es einige Kommandos, die den Cursor entsprechend des momentan sichtbaren Textausschnitts bewegen. Dieses wird desöfteren auch als *Scrollen* bezeichnet.

[5] In den meisten `vi` Versionen ist es möglich, die auf der Tastatur vorgesehenen Pfeiltasten zu benutzen. Dieses ist u.U. jedoch nur im Kommandomodus möglich !

Tastenfolge	Auswirkung
H	bewegt den Cursor zum **Anfang** des aktuellen Ausschnitts.
L	bewegt den Cursor zur **letzten Zeile** des aktuellen Ausschnitts.
M	bewegt den Cursor zur **Mitte** des aktuellen Ausschnitts.
CTRL + u	verschiebt den Ausschnitt um eine halbe Seite nach **oben**.
CTRL + d	verschiebt den Ausschnitt um eine halbe Seite nach **unten**.
CTRL + b	verschiebt den Ausschnitt um eine Seite **zurück**.
CTRL + f	verschiebt den Ausschnitt um eine Seite **vorwärts**.

Texteingabe Um eine Zeichenfolge in den aktuellen Text einzufügen, muß im Kommandomodus einer der folgenden Befehle eingegeben werden. Der Einfügemodus wird durch ESC beendet.

Tastenfolge	Auswirkung
i	geht in den Einfügemodus und fügt nachfolgende Zeichen **vor** die aktuelle Position des Cursors ein.
I	fügt den Text an den **Anfang der Zeile** ein.
a	geht in den Einfügemodus und fügt nachfolgende Zeichen **hinter** die aktuelle Position des Cursors ein.
A	fügt den Text ans **Ende** der aktuellen **Zeile**.
o	erzeugt eine neue Zeile **hinter** der aktuellen, geht in den Einfügemodus und fügt die nachfolgenden Zeichen an den Anfang der neuen Zeile ein.
O	erzeugt eine neue Zeile **vor** der aktuellen Zeile, ansonsten wie o.

Durch i wird vor die aktuelle Position des Cursors der nachfolgende Text eingefügt. Nach der Betätigung der ESC –Taste befindet sich der Editor wieder im Kommandomodus. Wurde bei der Eingabe ein Fehler gemacht, so muß zuerst in den Kommandomodus gewechselt, der Cursor gegebenfalls positioniert und anschließend die Korrektur durchgeführt werden. Es ist also nicht möglich, im Einfügemodus den Cursor zu positionieren[6]. Alle Tastatureingaben bis auf die ESC –Taste werden als Text und nicht als Kommando interpretiert.

Neben dem eigentlichen Einfügemodus ist es auch möglich, bereits bestehende Objekte zu ersetzen. Die nachfolgende Tabelle faßt die Möglichkeiten zusammen.

Tastenfolge	Auswirkung
R	die weitere Texteingabe **ersetzt** bereits vorhandene Zeichen (Überschreiben).
s	ersetzt das Zeichen unter dem Cursor durch den folgenden Text.
S	ersetzt die gesamte Zeile.

[6] Es sei angeraten zu überprüfen, ob die benutzte Version dieses doch ermöglicht.

Anzahl `c` *obj*	ersetzt *Anzahl* Objekte *obj* durch den nachfolgenden Text.
Anzahl `d` *obj*	löscht eine Anzahl *Anzahl* der Objekte *obj*.

Um also zum Beispiel in den folgenden Zeichenketten

```
Die objekt-orientierte Sprache C++
```

das `objekt-orientierte` zu löschen, ist zuerst im Kommandomodus der Cursor auf das erste `o` zu bewegen gefolgt von der Befehlssequenz `3` `d` `w`.
Letzteres ist äquivalent zu `d` `3` `w` bzw. kann in diesem Fall zu `d` `W` verkürzt werden.

Kopieren, Verschieben und Löschen Wichtige Operationen zum Editieren sind das Kopieren, Verschieben und Löschen von Quelltexten. Mit dem `c` und `d` Kommando lassen sich bereits die gängigsten Korrekturen ermöglichen.

Tastenfolge	Auswirkung
Anzahl `x`	**löscht rechts** vom Cursor *Anzahl* Zeichen.
Anzahl `X`	**löscht links** vom Cursor *Anzahl* Zeichen.
Anzahl `d` `d`	**löscht** einschließlich der aktuellen *Anzahl* Zeilen.

Der `vi`–Editor besitzt ingesamt 26 Puffer, die mit den Buchstaben `a` bis `z` angesprochen werden können. In den Löschpuffern `1` bis `9` befinden sich die Zeichenketten, die in den letzten neun Löschoperationen entfernt worden sind. Der Text, der beim jüngsten Löschen beseitigt wurde, steht dabei im Puffer `1`.

Um einen Puffer anzusprechen, ist die Referenzierung durch `"` möglich, d.h. `"` `a` bezeichnet den Puffer `a`.
Wird kein Puffername angegeben, so wird implizit immer der namenlose Puffer für die Operationen verwendet.

Tastenfolge	Auswirkung
buf `y` *Anzahl obj*	kopiert eine Anzahl *Anzahl* von Objekten *obj* in den Puffer *buf*.
buf Anzahl `y` `y`	kopiert eine Anzahl *Anzahl* von Zeilen in den Puffer *buf*.
buf `p`	fügt den Text aus Puffer *buf* **hinter** der aktuellen Cursorposition ein.
buf `P`	fügt den Text aus Puffer *buf* **vor** der aktuellen Cursorposition ein.

Beispiele:

`"` `a` `y` `W`

sichert das Wort, auf dem der Cursor sich gerade befindet in den Puffer `a`.

`"` `b` `P`

kopiert den Inhalt aus dem Puffer `b` vor die aktuelle Cursorposition.

Suchen und Ersetzen Im `vi`–Modus stehen nur eine beschränkte Anzahl von Such– bzw. Ersetzungsbefehlen zur Verfügung.

Tastenfolge	Auswirkung
`/` *string*	sucht vorwärts im Text nach *string* und bewegt den Cursor auf den Anfang des gefundenen *string*.
`?` *string*	sucht rückwärts im Text nach *string* und bewegt den Cursor auf den Anfang des gefundenen *string*.
`n`	wiederholt den letzten Suchvorgang.
`N`	wiederholt den letzten Suchvorgang in umgekehrter Richtung.

Hingegen bietet der `ex` einen mächtigen Befehlsvorrat, der (fast) allen Such– und Ersetzungswünschen gerecht wird.

Tastenfolge	Auswirkung
`:` `g` `/` *key* `/` com	**sucht** im Text alle Zeilen, die das Suchwort *key* enthalten, markiert und führt dann auf diesen die angegebenen `ex`–Befehle *com* aus.
`:` `s` `/` *text1* `/` *text2* `/` *opt*	**ersetzt** auf der **aktuellen** Zeile den Text *text1* durch die Zeichenkette *text2*. Als Optionen *opt* sind u.a. möglich: `g` ersetzt alle Vorkommen in der Zeile und `c` fragt nach, ob ersetzt werden soll.

Als Suchwörter sind sogenannte **reguläre Ausdrücke** zugelassen. Ist die Zeichenkette *text2* leer, so wird keine Ersetzung durchgeführt, sondern *text1* gelöscht.
Den `ex`–Befehlen ist es möglich, einen Bereich anzugegeben, in dem diese ausgeführt werden können. Ansonsten wirkt der Befehl nur auf einer Zeile.

Beispiel:

`:` `s` `/` `int` `/` `long` `/`

In der aktuellen Zeile wird die erste Zeichenkette `int` durch die Zeichenkette `long` ersetzt.

`:` `s` `/` `int` `/` `long` `/` `g`

In der aktuellen Zeile werden alle Zeichenketten `int` durch eine Zeichenkette `long` ersetzt.

`:` `1` `,` `$` `s` `/` `int` `/` `long` `/` `g`

In dem Bereich von der ersten bis zur letzten Zeile[7] werden alle Zeichenketten `int` in einer Zeile durch die Zeichenkette `long` ersetzt.

Es sei nochmal angemerkt, daß sämtliche `ex`–Befehle mit einem `RETURN` abgeschlossen werden müssen.

Sonstiges und Nützliches Weiterhin sind die folgenden Kommandos nützlich. Insbesondere sei auf das Kommando `%` hingewiesen, welches es bei der Quelltexteingabe ermöglicht, eine korrekte Klammerung zu überprüfen.

Tastenfolge	Auswirkung
`%`	steht der Cursor auf einer geöffneten `(`, `{` oder `[` Klammer, so wird dieser auf die entsprechende geschlossene Klammer bewegt; das umgekehrte gilt, wenn sich der Cursor zu Beginn auf einer geschlossenen Klammer befindet.

[7] Die Bedeutung des Dollarzeichens $ unterscheidet sich zum Kommandomodus.

Tastenfolge	Auswirkung
`.`	wiederholt das letzte Änderungskommando.
`u`	hebt die letzte Änderung auf (**undo**).
`U`	hebt die letzten Änderungen in der aktuellen Zeile auf.
`CTRL` + `l`	gibt den Bildschirm neu aus.
`CTRL` + `g`	zeigt den Namen der aktuellen Datei, die Gesamtzahl der Zeilen sowie die Zeile des Cursors an.
Anzahl `J`	entfernt das Neue–Zeile Zeichen in der aktuellen Zeile, d.h. die nachfolgende Zeile wird an die aktuelle angehängt. Dieses wird *Anzahl* mal wiederholt.
`<` *obj*	**schiebt** das Objekt *obj* um acht Positionen **nach links**.
`>` *obj*	**schiebt** das Objekt *obj* um acht Positionen **nach rechts**.

Datei–Kommandos Abschließend sind die Kommandos aufgeführt, mit denen Dateien in den `vi` eingelesen und gesichert werden können. Diese werden im `ex`–Modus angenommen, d.h. gegebenfalls muß zuerst in den Kommando–Modus gewechselt werden (durch `:` wird für die folgende Zeile vom Kommando– in den `ex`–Modus gewechselt).

Tastenfolge	Auswirkung
`:` `w` *name*	speichert den aktuellen Text ab. Optional kann ein Name *name* für eine neue Datei angegeben werden.
`:` `w` `!` *name*	speichert den aktuellen Text und überschreibt gegebenfalls ohne Nachfrage.
`:` `w` `q`	speichert den Text und verläßt `vi`.
`:` `q` `!`	verläßt `vi` **ohne** voher den Text zu sichern.
`:` `e` *name*	liest die angegebene Datei *name* ein. Zuvor muß die aktuelle Datei gesichert werden.
`:` `e` `!` *name*	sichert die aktuelle Datei und liest anschließend die Datei *name* ein.
`:` `r` *name*	fügt die Datei *name* an die aktuelle Position des Cursor ein.
`:` `r` `!` *com*	führt den angegebenen UNIX™–Befehl *com* aus und fügt dessen Ausgabe in den Text an die aktuelle Stelle ein.

An dieser Stelle konnte nur ein ganz kleiner Ausschnitt der Möglichkeiten, die `vi` bietet, betrachtet werden. Für eine ausführlichere und weiterführende Diskussion sei auf die angegebene Literatur verwiesen.

Im folgenden Abschnitt wird gezeigt, wie der `vi` benutzt werden kann, um den Programmiervorgang (im wesentlichen die Quelltexteingabe) zu vereinfachen.

2.2.1.2 Anpassungen

Mit Hilfe des mächtigen Befehlsumfanges ist es möglich, den `vi`–Editor zur Vereinfachung der Quelltexteingabe einzusetzen.

Beim Start des Editors wird die Datei `~/.exrc` [8] eingelesen und es werden die darin enthaltenen Kommandos ausgeführt. Somit ist es möglich, die Anpassungen nur einmal vorzunehmen, sofern diese sich von Sitzung zu Sitzung nicht unterscheiden.
Kommentarzeilen werden übrigens entgegen aller Konventionen in dieser Datei nicht mit einem # eingeleitet, sondern mit dem " Zeichen in der ganz linken Spalte.

Interne Optionen Der `vi` bzw. `ex`–Editor kann durch die Modifikation einiger interner Parameter an die individuellen Bedürfnisse angepaßt werden.

Tastenfolge	Auswirkung
`:` `s` `e` `t` *opt* `=` *wert*	setzt die Option *opt* auf den numerischen Wert *wert*. Es dürfen keine Leerzeichen das Gleichheitszeichen umgeben.
`:` `s` `e` `t` *opt wert*	setzt die Option *opt* auf den Wert *wert*. Die Option darf nur zwei zulässige Werte besitzen, nämlich *wert* und `no`*wert*.
`:` `s` `e` `t` ` ` `a` `l` `l`	gibt alle Optionen mit den aktuellen Werten aus.

[8] Das Tildezeichen bezeichnet in UNIX™–ähnlichen Systemen das persönliche Verzeichnis (HOME) des Benutzers. In anderen Betriebssystemen ist dieses oftmals nicht gegeben!

Die nachfolgende Tabelle in Abbildung 2.2 listet einige wichtige Optionen auf.

Makros Der `vi`–Editor unterstützt zwei Möglichkeiten die Eingabe wiederkehrender Befehlssequenzen oder Textbausteine zu vereinfachen:

Abkürzungen werden im Einfügemodus erkannt und zu einer vorher definierten Langform expandiert.

Makros sind Zusammenfassungen von Befehlssequenzen, die einer bestimmten Taste zugewiesen und im Kommandomodus ausgeführt werden.

Die Definition eines Kürzels kann mit den folgenden Kommandos erfolgen.

Tastenfolge	Auswirkung
`:` `a` `b` *kurz lang*	definiert *kurz* als Abkürzung der Langform *lang*.
`:` `u` `n` `a` *kurz*	hebt eine vorher definierte Abkürzung *kurz* wieder auf.

Beispiel:

`:` `a` `b` `UniHi Universitaet Hildesheim`

Wird im Einfügemodus nun der Text `UniHi` eingegeben, ersetzt der `vi`–Editor automatisch dieses durch `Universitaet Hildesheim`. Dieses wird jedoch nur dann durchgeführt, wenn dem Kürzel ein nicht alphanumerisches Zeichen, z.B. ein Leerzeichen, folgt.

Parameter	Voreinstellung	Bedeutung
`shiftwidth`	`8`	gibt die Einrücktiefe bei den `<` und `>` Befehlen an (siehe oben).
`ignorecase`	`noic`	unterscheidet bei Suchbefehlen nicht zwischen Groß– und Kleinbuchstaben.
`number`	`nonu`	gibt bei der Darstellung des editierten Textes eine entsprechende Zeilennummer aus.
`autoindent`	`noai`	rückt jede neue Zeile gleich weit der vorherigen ein. Ansonsten beginnt jede Zeile in der ganz linken Spalte.
`showmatch`	`noshowmatch`	wird im Einfügemodus eine geschlossene Klammer `)` oder `}` eingegeben, springt der Cursor kurzzeitig auf die entsprechende geöffnete Klammer und zurück.
`showmode`	`noshowmode`	gibt den aktuellen Arbeitsmodus (z.B. Einfügemodus) in der Statuszeile aus.

Abbildung 2.2: Interne Optionen des `vi`

Die zweite Möglichkeit ist es, Makros zu definieren. Hierzu können die folgenden `ex`–Kommandos benutzt werden.

Tastenfolge	Auswirkung
`:` `m` `a` `p` *key com*	belegt die Taste *key* mit der Befehlssequenz *com*.
`:` `u` `n` `m` `a` `p` *key*	hebt die Makrodefinition *key* wieder auf.

Die meisten Tasten haben im `vi` Kommandomodus bereits eine vordefinierte Funktion. Daher bietet es sich an, die auf den meisten Tastaturen vorhandenen Funktionstasten mit einem Makro zu belegen.

Es stellt sich nun die Frage, wie man bei der Makrodefinition eine nicht alphanumerische Taste (z.B. `F 10`) eingeben kann, ohne daß der Tastendruck als Kommando interpretiert wird.

Hierzu dient im Einfügemodus und bei der Eingabe in der Statuszeile die Tastenfolge `CTRL` + `v`, die dafür sorgt, daß die nachfolgende Taste als Text interpretiert wird. Die nicht alphanumerischen Tasten, wie z.B. `RETURN`, `F 3` oder `ESC` werden als sogenannte **Escape–Sequenzen**[9] dargestellt; zum Beispiel `RETURN` als `^M`.

Um die Taste `CTRL` + `l` mit dem UNIX™ Kommando `ls -al` zu belegen, ist folgende Makrodefinition nötig:

`:` `m` `a` `p` ` ` `CTRL` + `v` `CTRL` + `l` `!ls -al`

In der Statuszeile wird hingegen folgendes angezeigt:

[9] Die Escape–Sequenzen sind nicht auf allen UNIX™ Systemen gleich. Das Neue–Zeile Zeichen belegt den ASCII Code 13 und wird daher mit `^M` dargestellt. Die Funktionstasten werden jedoch unterschiedlich behandelt: auf einem SUN System wird zum Beispiel die Funktionstaste F8 als `^[[231z` dargestellt, hingegen unter Linux mit einer PC MF2 Tastatur als `^[[19`. Die in dieser Fibel abgedruckten Escape–Sequenzen können daher von denen der eigenen Tastatur abweichen!

`:map ^L !ls -al`

Falls keine Funktionstasten zur Verfügung stehen, kann man Pseudo–Funktionstasten einführen. Diese werden im Kommandomodus mit [#] *char* eingegeben, wobei *char* ein Buchstabe oder eine Ziffer ist.
Beispiel:

[:][m][a][p][ ][#][l] `!ls -al`

Natürlich ist es auch möglich, die Tasten mit immer wiederkehrenden Textbausteinen zu belegen. Ein entsprechender Eintrag in der `.exrc`–Datei könnte zum Beispiel wie folgt aussehen:

```
map #c oclass {^M^M protected :
^M^Mpublic :^M^M};^[
```

Wird im Kommandomodus nun die Pseudo–Funktionstaste [#] [c] gedrückt, so wird in der dem Cursor nachfolgenden Zeile ein Klassendefinitionsformular eingefügt, welches dann um die entsprechenden Methoden, Klassenname usw. ergänzt werden kann.
Diese Möglichkeiten lassen sich natürlich beliebig erweitern und verfeinern.

In [Sto95] wird gezeigt, wie der `vi`–Editor unter Benutzung der Makrobefehle an eigene Programmierbedürfnisse angepaßt werden kann. Fernerhin werden einige Shell–Skripte angegeben, die es ermöglichen, automatisch Funktionsprotoypen zu generieren.

`ctags` Der letzte Abschnitt zum `vi` behandelt ein kleines zusätzliches Werkzeug, welches es ermöglicht, in auf mehreren Dateien verteilten Quelltexten zu suchen.
So sucht man oftmals nach der Stelle in den Quellen, an der zum Beispiel eine Variable oder Klasse definiert worden ist. Das `ctags` Werkzeug ermöglicht es, kontextsensitiv und automatisch die richtige Datei und Zeile zu zeigen.

Das `ctags` Programm[10] durchsucht die angegebenen Quelltextdateien nach bestimmten Begriffen (*tags*) und erzeugt eine Datei `tags`, die Zeileneinträge mit folgendem Format hat:

Begriff *Datei* *Kommando*

Die Begriffe sind aufsteigend alphabetisch sortiert.
Wird nun nach einem Begriff *tag* gesucht, so untersucht der `vi`–Editor die spezifizierten tags–Dateien nach dem gesuchten Schlüsselbegriff. Wird ein entsprechender Eintrag gefunden, so wird anschließend in der dem Begriff folgenden Datei das nachstehende Kommando ausgeführt. Dieses sind i.allg. die weiter oben beschriebenen Suchkommandos.
Die nachfolgende Tabelle faßt die nötigen Befehle zur Benutzung des `ctags`–Mechanismus zusammen.

Tastenfolge	Auswirkung
[:][t][a][g] *beg*	sucht in den tags–Dateien nach dem Begriff *beg* und plaziert gegebenfalls den Cursor.

[10] `ctags` versteht nicht nur C bzw. C++ Syntax, sondern kann auch dazu benutzt werden, LaTeX, Pascal oder Fortran Quelltexte nach Suchbegriffen sortiert aufzubereiten. Der Typ des Quelltextes wird automatisch anhand der Dateiendung erkannt. Bei der GNU Variante `ctags` kann durch den Schalter `-C` erzwungen werden, daß Dateien mit den Endungen `.h` und `.c` explizit als C++ Quelltexte behandelt werden.

CTRL +]	sucht in den tags–Dateien nach dem Begriff, welches sich an der aktuellen Cursorposition befindet und plaziert gegebenfalls den Cursor.

Des weiteren sind die folgenden (vgl. Abbildung 2.3) zwei internen Optionen wichtig, die mit dem s e t Kommandos angepasst werden können.
Möchte man nun nach einem bestimmten Begriff suchen, ist es zum einen möglich, den Cursor auf das erste Zeichen im Quelltext zu positionieren und dann anschließend CTRL +] zu drücken.
Die zweite Möglichkeit ist es, den Suchbegriff explizit mit dem t a g Kommando einzugeben.
In beiden Fällen durchsucht der Editor die angebenen tags–Dateien und positioniert den Cursor, wenn die Suche erfolgreich verlaufen ist.

Natürlich ist es auch möglich, die `tags` Datei manuell zu erstellen oder zu verändern. So kann zum Beispiel mit nachfolgenden Einträgen[11] eine Online–Hilfe realisiert werden.

```
gcc     %      !man gcc
make    %      !man make
```

Aufgabe 2.1
Machen Sie sich mit der Bedienung des `vi`–Editors vertraut.

a Wie lassen sich die Objekte Satz, Absatz und Abschnitt neu definieren ?

b Wie lauten die Kommandos im `vi`–Modus, um innerhalb einer Zeile zu suchen ?

c Wie werden bestimmte Stellen markiert ? Wie ist es möglich, mit dem Cursor zu solchen Marken hinzuspringen ?

d Welche Funktionalität hat das `ex`–Kommando : @ *buf*, wobei *buf* einen der 26 Textpuffer bezeichnet ?

e Definieren Sie Makros mit den Funktionen *Markieren*, *Ausschneiden* und *Kopieren* von Textstellen.

2.2.2 Der Editor `emacs`

Das Programm `emacs` ist kein Editor im üblichen Sinn. `emacs` ist vielmehr eine komplette Schnittstelle zwischen Benutzerin bzw. Benutzer und Rechner. `emacs`

- ist ein Lisp Interpreter System,
- ermöglicht es, elektronische Post zu lesen, schreiben und zu versenden,
- ist ein vollständig programmierbarer Editor,
- „versteht“ u.a. C, C++, Quelltexte,
- ersetzt die übliche UNIX™ Shell,
- ist ein komfortables Hilfe– und Informationssystem[12],
- formatiert Texte automatisch,
- besitzt eine integrierte Schnittstelle zum Debugger,
- ist frei verfügbar[13],

[11] Das Prozentzeichen % hat eine besondere Bedeutung und bezeichnet die momentan geöffnete Datei.

[12] Die bereits in Abschnitt 1.3 beschriebene Online Hilfe ist ein Teil vom `emacs`.

[13] unter GNU Bedingungen

Parameter	Voreinstellung	Bedeutung
tags	tags /usr/lib/tags	bestimmt die zu durchsuchenden tags–Dateien. Es können mehrere durch Komma getrennte Dateien angegeben werden.
taglength	0	gibt die Anzahl der signifikanten Zeichen beim Durchsuchen an (bei 0 sind alle Zeichen signifikant).

Abbildung 2.3: tags Optionen des vi

- ...

Hier ist nur ein kleiner Ausschnitt aus der gesamten möglichen Funktionalität aufgeführt. Es zeigt sich, daß eine Beherrschung sämtlicher Kommandos eine gewisse Zeit in Anspruch nimmt. Die folgenden Abschnitte geben eine kurze Einführung in den emacs Editor; insbesondere wird beschrieben, wie dieser als integrierte Entwicklungsumgebung benutzt werden kann.

Der emacs ist von der ursprünglichen PDP–10 Version, entworfen von **Richard Stallman**, auf verschiedene Plattformen portiert worden. Auch existieren neben verschiedenen Versionen auch unterschiedliche Ausbaustufen. Die hier gemachten Angaben beziehen sich auf den GNU emacs[14] mit X–Window Unterstützung. Die Funktionen sind in der Regel identisch mit der Version ohne X11, das Erscheinungsbild ist ohne X11–Option jedoch einfacher. Des weiteren gibt es den xemacs[15], der eine wesentlich verbesserte Anpassung an die X–Oberfläche und eine erweiterte Funktionalität bietet.

2.2.2.1 Allgemeines

Neben vi ist der Editor emacs die Standardeingabemöglichkeit für Texte in einem UNIX™ System. Anders als bei vi wird nicht zwischen Einfüge– und Kommandomodus unterschieden. Statt dessen werden sämtliche Befehle durch zwei spezielle Tasten, nämlich CTRL und ESC, eingeleitet.

Abbildung 2.4 zeigt den GNU emacs mit X–Window Unterstützung. emacs besitzt ein eigenes Fenstersystem. Dieses ist bei der emacs Variante ohne X11–Option besonders vorteilhaft, wenn nicht mit einer grafischen Bedienoberfläche gearbeitet werden kann.
emacs arbeitet ähnlich wie vi mit **Puffer**n, das heißt Dateien werden zuerst in einen temporären Textspeicher geladen und dann (eventuell) verändert. Erst durch einen expliziten Schreibbefehl wird der Puffer wieder zurückgeschrieben. Jeder Puffer erhält i.allg. den Namen der zuvor gelesenen Datei.
Der emacs kann sich in verschiedenen Zuständen oder Modi befinden. Der Editier–**Modus** bestimmt die Funktionalität des emacs. Jeder Modus[16] besitzt eine bestimmte Grundfunktionalität, die die gängigen Kommandos, z.B. zum Löschen, Kopieren, Datei lesen und schreiben, umfaßt. Dazu kommen ergänzende Befehle, die nur in bestimmten Situationen sinnvoll erscheinen. So gibt es z.B. einen **C Modus**, in dem programmierspezifische Aktionen möglich werden. Im nächsten Abschnitt wird darauf eingegangen.

[14] Version 19.28.1
[15] ehemals Lucid emacs
[16] Genau genommen setzt sich ein Modus aus **Hauptmodus** und **Untermodus** zusammen.

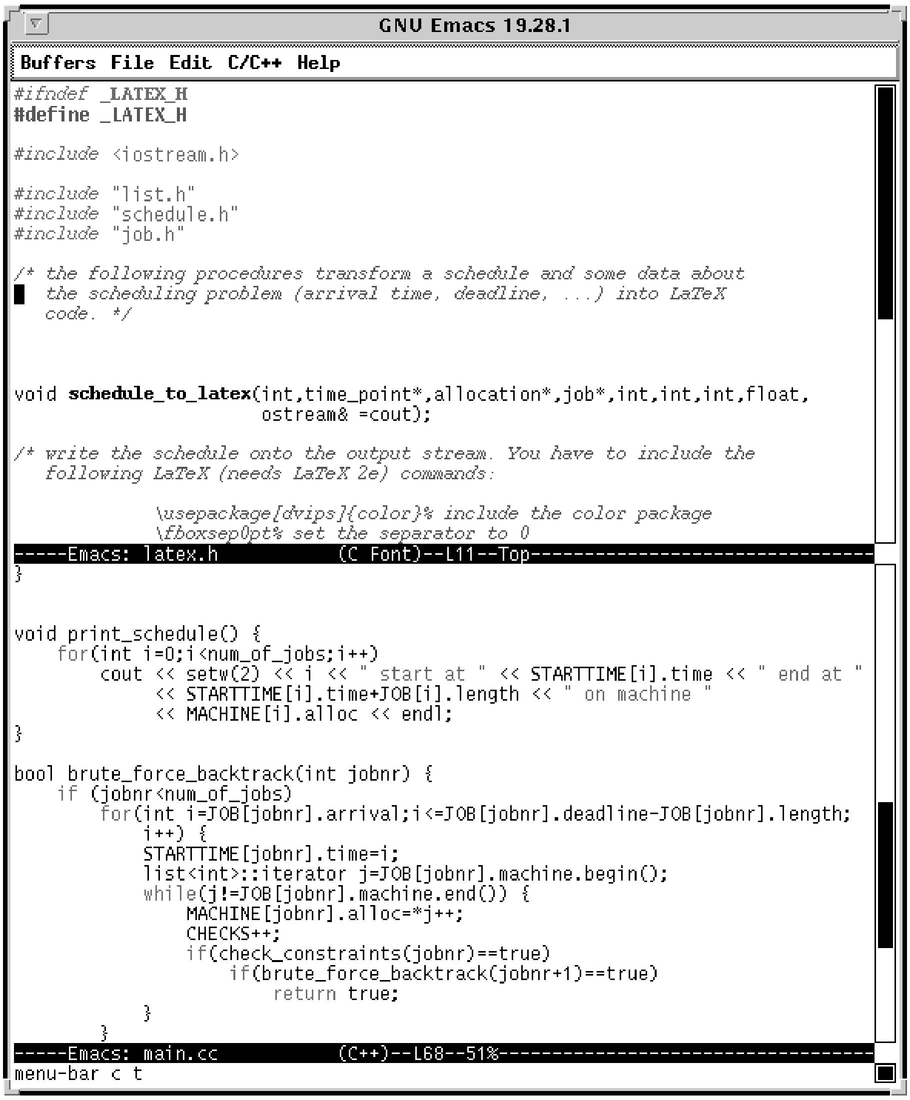


Abbildung 2.4: Der GNU `emacs` unter X11

Der emacs besitzt ein eigenes Fenstersystem. Jedes Fenster besitzt eine untere Statuszeile, in der der aktuelle Puffername, der Modus und die prozentuale Position des angezeigten Ausschnittes des Puffers aufgeführt ist (Abbildung 2.4 zeigt zwei Fenster). Zwei Sterne am linken Rand zeigen an, daß der Puffer seit dem letzten Sichern modifiziert wurde.
Die letzte Zeile wird als **Minipuffer** bezeichnet, in dem Statusmeldungen angezeigt und Befehle eingegeben werden. Nach Eingabe von CTRL bzw. ESC wird automatisch in diesen gewechselt.
In einem Fenster selber wird der Inhalt des angewählten Puffers angezeigt. emacs wechselt beim Einlesen von speziellen Dateien, so z.B. bei C und C++ Quelltexten, in einen entsprechenden Modus und bereitet die Datei auf. So werden zum Beispiel Schlüsselwörter in bestimmten Farben angezeigt.
Wird bei der Eingabe über das Zeilenende hinweg geschrieben, so wird der Text automatisch umgebrochen.

Einige Befehle sind auf Objekte anzuwenden. Ein **Wort** beginnt an der aktuellen Position des Cursors und endet beim ersten Zeichen, welches kein Buchstabe oder keine Ziffer ist. Ein **Satz** endet mit einem Punkt als Satzendezeichen und zwei anschließenden Leerzeichen. Ein **Absatz** ist eine Folge von Sätzen, die mit einem Tabulatorzeichen, mindestens einem Leerzeichen oder mindestens einer Leerzeile beginnt[17].

Die wichtigsten Befehle sind über eine kurze Sequenz von Tasten zu erreichen. Andere müssen explizit in Langform angegeben werden. Die Kurzform

CTRL + f

ist äquivalent zu

ESC x forward-char RETURN [18].

Die hier eingeschlagene Beschreibung des emacs folgt im wesentlichen dem Buch [CR91] von Debra Cameron und Bill Rosenblatt , in dem die interessierte Leserin bzw. der interessierte Leser weitere und ausführliche Informationen erhält.

Pufferkommandos Wie eingangs geschildert, arbeitet emacs mit Puffern und Fenstern. Jedes Fenster zeigt genau einen Puffer an. Es ist möglich, einen Puffer in mehr als einem Fenster anzeigen zulassen.
Die folgende Tabelle zeigt die möglichen Kommandos, um zwischen Puffern und Fenstern zu wechseln.

Tastenfolge	Auswirkung
CTRL + x b	wechselt im aktuellen Fenster den spezifizierten Puffer.
CTRL + x s	sichert alle Puffer.
CTRL + x k	löscht einen Puffer.
CTRL + x CTRL + b	öffnet ein Fenster und zeigt sämtliche Puffer an.

Der emacs besitzt die Möglichkeit der automatischen Befehls– bzw. Dateinamen–Ergänzung (*file and command completion*), die auch einige UNIX™ Shells bieten. Durch die Eingabe der TAB Taste versucht emacs, soweit möglich, die bisher eingegeben Zeichen zu einem gültigen Datei– oder Befehlsnamen zu ergänzen. Dieses ist immer dann möglich, wenn eine Eingabe im Minipuffer erforderlich ist.
Mit CTRL + x b wird ein gewählter Puffer geöffnet. Falls der Puffer nicht existiert, wird

[17] Natürlich können die beiden Objektdefinitionen modifiziert werden. Die notwendige interne Variable heißt sentence-end bzw. paragraph-end.

[18] Zu beachten ist, daß die CTRL Taste gedrückt sein muß, wenn anschließend ein Buchstabe oder eine Ziffer folgt. Hingegen ist die ESC Taste vor der weiteren Eingabe loszulassen.

dieser neu angelegt. Um einen so neu angelegten Puffer abzuspeichern, muß dieser mit `CTRL` + `x` `CTRL` + `w` einen Namen zugewiesen bekommen (vgl. Abschnitt über Dateikommandos). Der Puffername muß nicht zwingend mit dem Dateinamen übereinstimmen.

Unbeabsichtigtes Löschen von Dateien bzw. Puffern ist i.allg. nicht möglich, da der `emacs` im Zweifelsfall nachfragt[19].

Durch den `CTRL` + `x` `CTRL` + `b` Befehl wird, soweit nicht bereits geschehen, ein neuer Puffer `*Buffer List*` erzeugt und in einem eigenen Fenster angezeigt. Dieser Puffer umfaßt eine Tabelle, in der einige Informationen (Größe, Modus und Dateiname) zu einem jedem Puffer aufgeführt sind.

Im `*Buffer List*` Fenster sind die folgenden Befehle möglich. Die Eingabe erfolgt in der ganz linken Spalte einer entsprechenden Zeile.

Taste	Auswirkung
`d`	markiert den Puffer zum Löschen.
`s`	markiert den Puffer zum Sichern.
`u`	hebt eine Markierung wieder auf.
`x`	führt die Aktionen auf markierte Puffer aus.
`1`	zeigt den angewählten Puffer in einem Fenster an.
`2`	zeigt den angewählten und den folgenden Puffer in zwei Fenstern an.
`f`	ersetzt das `*Buffer List*` Fenster mit dem angewählten Puffer.

Die beiden `d` und `s` Kommandos werden erst nach Eingabe von `x` ausgeführt; die anderen im Gegensatz dazu sofort.

[19] Ein Prinzip des `emacs` ist es, *kritische* Entscheidungen mit `yes` oder `no`, *weniger kritische* mit `y` oder `n` zu bestätigen.

Neben der Möglichkeit, die Puffer zu wechseln, sind verschiedene Kommandos vorgesehen, um das Arbeiten mit den verschiedenen Fenstern zu vereinfachen.

Tastenfolge	Auswirkung
`CTRL` + `x` `2`	teilt das aktuelle Fenster horizontal.
`CTRL` + `x` `o`	wechselt in das nächste Fenster.
`CTRL` + `x` `0`	löscht das aktuelle Fenster.
`CTRL` + `x` `1`	löscht alle Fenster bis auf das aktuelle.
`CTRL` + `x` `4` `f`	liest in das zweite Fenster eine Datei ein.
`CTRL` + `x` `4` `b`	wechselt im zweiten Fenster den Puffer.

Wird ein Fenster gelöscht, so bleibt der entsprechende Puffer erhalten. Wird hingegen ein Puffer gelöscht, so verschwindet auch das entsprechende Fenster.

Cursorbewegung Im allgemeinen ist es möglich, die speziell ausgewiesenen Pfeiltasten zur Bewegung des Cursors zu benutzen.

Tastenfolge	Auswirkung
`CTRL` + `f`	bewegt den Cursor ein Zeichen nach rechts.
`CTRL` + `b`	bewegt den Cursor ein Zeichen nach links.
`CTRL` + `p`	bewegt den Cursor eine Zeile nach oben.
`CTRL` + `n`	bewegt den Cursor eine Zeile nach unten.

CTRL + a	bewegt den Cursor zum ersten sichtbaren Zeichen der aktuellen Zeile.
CTRL + e	bewegt den Cursor zum letzten sichtbaren Zeichen der aktuellen Zeile.
CTRL + v	bewegt den aktuellen sichtbaren Ausschnitt eine Bildschirmseite weiter.
ESC v	bewegt den aktuellen sichtbaren Ausschnitt eine Bildschirmseite zurück.
ESC f	bewegt den Cursor ein Wort weiter.
ESC b	bewegt den Cursor ein Wort zurück.

Befehle können mit einem Wiederholungsfaktor versehen werden. Mit ESC *n* wird der nachfolgende Befehl *n* mal wiederholt. Also

ESC 10 CTRL + f

bewegt den Cursor 10 Stellen nach rechts. Ähnlich verhält sich der CTRL + u Befehl. Wird dieser ohne Argument *n* aufgerufen, wird der nachfolgende Befehl viermal ausgeführt. Mit

CTRL + u CTRL + u CTRL + f

wird der Cursor 16 Zeichen nach rechts bewegt.

Kopieren, Verschieben und Löschen Die nachfolgende Tabelle listet die häufigsten Kommandos zum Löschen auf. Wie bereits anfangs beschrieben ist es möglich, jedem Kommando einen Wiederholungsfaktor voranzustellen. So löschen die folgenden Anweisungen

CTRL + u 5 CTRL + d

fünf nachfolgende Zeichen beginnend bei der aktuellen Cursorposition.

Tastenfolge	Auswirkung
CTRL + d	löscht den Buchstaben unter dem Cursor.
ESC + DEL	löscht das vorangehende Wort.
ESC + d	löscht das Wort unter dem Cursor.
CTRL + k	löscht den Text vom Cursor bis zum Zeilenende.
ESC + k	löscht den Text vom Cursor bis zum Satzende.

Neben den reinen Löschfunktionen sind insbesondere Befehle zum markieren, kopieren und bewegen von Textbereichen relevant.
Bei der `emacs` Version mit X–Window Unterstützung können diese Befehle auch in der Menüleiste abgerufen werden.

Tastenfolge	Auswirkung
CTRL + ␣	markiert den Beginn einer Region.
CTRL + @	dito.
CTRL + x h	markiert den ganzen Puffer.
CTRL + x CTRL + x	bewegt den Cursor zum Anfang bzw. zum Ende des markierten Bereichs.
CTRL + y	fügt den Text an die aktuelle Position des Cursors ein.

Um einen bestimmten Abschnitt zu löschen, muß der entsprechende Bereich markiert werden. Hierzu wird der Cursor auf den Beginn gesetzt und markiert. Anschließend ist der Cursor an das Ende des zu löschenen Bereichs zu bewegen. Mit `CTRL` + `w` wird der markierte Bereich gelöscht, mit `ESC` + `w` in einen Löschpuffer kopiert. Der gelöschte bzw. kopierte Text kann dann mit `CTRL` + `y` an der aktuellen Cursorposition eingefügt werden.
Eine besondere Funktion besitzt der `ESC` `y` Befehl. Hiermit ist es möglich, den mit `CTRL` + `y` eingefügten Text durch den in den letzten Löschoperationen entfernten Text zu ersetzen. Hierbei wird beim jüngst gelöschten Text begonnen.
Das folgende Beispiel zeigt, wie diese eingesetzt werden. Mit den letzten Löschoperationen sind einzeln die Wörter `orange`, `blau`, `rot` und `gelb` entfernt worden. Der Cursor befinde sich am Ende des unten aufgeführten Textes. Aus

Die Farbe des Autos ist .

wird dann bei Ausführung von

`CTRL` `y`

Die Farbe des Autos ist gelb.

`ESC` `y`

Die Farbe des Autos ist rot.

`ESC` `y`

Die Farbe des Autos ist blau.

`ESC` `y`

Die Farbe des Autos ist orange.

Suchen und Ersetzen Die bisherigen Kommandos ermöglichen es schon, Texte zu editieren. Aber erst durch **Suchkommandos** bzw. **Such– und Ersetzungskommandos** wird ein Editor zu einem effizienten Werkzeug.
Auch an dieser Stelle kann nur ein kurzer Überblick auf die `emacs` Befehle gegeben werden. Es sei abermals auf die angegebene Literatur bzw. auf die Online–Hilfe verwiesen.

Tastenfolge	Auswirkung
`CTRL` + `s`	startet die Vorwärtssuche.
`CTRL` + `r`	startet die Rückwärtssuche.

Bei den hier aufgeführten Suchkommandos[20] handelt es sich um eine sogenannte **inkrementelle Suche**, d.h. der Editor startet die Suche im aktuellen Puffer, sobald ein Zeichen eingegeben wird.
Ist eine einfache, d.h. nicht inkrementelle Suche gewünscht, so muß vor und nach der Eingabe der Suchzeichenkette ein `RETURN` eingefügt werden. Durch `CTRL` + `s` wird auf das nächste Vorkommen mit dem Cursor gesprungen.

`CTRL` + `r` `RETURN` `emacs` `RETURN`

sucht nach der Eingabe des zweiten Returns rückwärts nach der Zeichenkette `emacs`.

Der Such– und Ersetzungsbefehl ist hingegen ein wenig komplizierter. Um diesen zu starten, ist die folgende Eingabe[21] nötig:

`ESC` `x` `replace-string` `RETURN`

Anschließend ist die zu ersetzene Zeichenkette und der Ersetzungstext jeweils mit einem `RETURN` als Abschluß einzugeben. Also zum Beispiel nach der Eingabe des obigen Befehls

`Hochschule` `RETURN` `Uni` `RETURN`

[20] `emacs` unterscheidet nicht zwischen Groß– und Kleinbuchstaben bei der Suche, d.h. er behandelt z.B. bei der Suche nach `for` die Zeichenketten `For`, `FOR` und `for` gleich. Bei einer Ersetzung bleibt die Schreibweise jedoch erhalten, d.h. bei dem angeführten Beispiel wird bei einer Ersetzung mit `tor` aus den verschiedenen Zeichenketten `Tor`, `TOR` bzw. `tor`. Ist dieses Verhalten nicht gewünscht, so ist die Variable `case-fold-search` von `t` auf `nil` zu setzen.

[21] Es sei an dieser Stelle nochmals angemerkt, daß `emacs` durch Eingabe der `TAB` Taste versucht, den bisher eingegebenen Befehl soweit wie möglich zu vervollständigen.

Ein etwas komfortableres und mächtigeres Kommando ist das **Such– und Ersetzungskommando mit Nachfrage**. Dieses wird mit `ESC` `%` gestartet. Nacheinander sind zu ersetzener und Ersetzungstext im Minipuffer einzugeben. `emacs` startet den Ersetzungsvorgang und plaziert den Cursor auf die erste zutreffende Zeichenkette. Bevor ersetzt wird, wartet `emacs` auf eine weitere Angabe der Benutzerin bzw. des Benutzers (Nachfrage). Unter anderem sind die folgenden Anweisungen erlaubt.

Taste	Auswirkung
`y`	ersetzt das aktuelle Vorkommen und fährt fort.
`n`	übergeht das aktuelle Vorkommen und fährt fort.
`.`	ersetzt das aktuelle Vorkommen und beendet den Ersetzungsvorgang.
`!`	ersetzt die restlichen Vorkommen ohne weitere Nachfragen.
`ESC`	beendet den Ersetzungsvorgang ohne das aktuelle Vorkommen zu ersetzten.

Sonstiges und Nützliches Der `emacs` besitzt eine Fülle von Befehlen. Die nachfolgende Tabelle listet einige weitere auf, die einem u.U. hilfreich sein können.

Tastenfolge	Auswirkung
`CTRL` + `h` `t`	startet eine Einführung in den `emacs`.
`CTRL` + `x` `u`	macht den letzten Befehl rückgängig.
`CTRL` + `l`	baut den Bildschirm neu auf.
`CTRL` + `g`	bricht den momentanten Befehl ab.
`CTRL` + `t`	vertauscht zwei Buchstaben.
`CTRL` + `T`	vertauscht zwei Wörter.
`CTRL` + `x` `CTRL` + `t`	vertauscht zwei Linien.

Eine nette Eigenschaft des `emacs` ist der **dired Modus**, eine Art Dateimanager, mit dem es möglich ist, Verzeichnisse anzuzeigen, Dateien zu kopieren oder umzubennen und vieles mehr. Um in diesen zu gelangen, reicht es, mit `CTRL` + `x` `CTRL` + `f` ein Verzeichnis einzulesen (vgl. auch den nächsten Absatz). Fernerhin ist es möglich, mit dem `ESC` `x` `dired` `RETURN` Befehl den dired Modus direkt zu aktivieren.

Dateikommandos Die nachfolgenden Kommandos zum Einlesen und Schreiben von Dateien stehen in engen Zusammenhang mit den bereits beschriebenen Pufferbefehlen. Die folgende Tabelle faßt diese zusammen.

Tastenfolge	Auswirkung
`CTRL` + `x` `CTRL` + `f`	liest die spezifizierte Datei ein.
`CTRL` + `x` `CTRL` + `v`	ersetzt im aktuellen Puffer durch eine alternative Datei.
`CTRL` + `x` `i`	fügt Datei an die Cursorposition ein.

CTRL + x CTRL + w	schreibt den aktuellen Puffer in eine Datei.
CTRL + x CTRL + s	sichert den aktuellen Puffer.
CTRL + x CTRL + c	sichert alle Puffer und verläßt `emacs`.

Der Unterschied zwischen CTRL + x CTRL + w und CTRL + x CTRL + s liegt darin, daß die Benutzerin bzw. der Benutzer explizit einen Namen der Datei angeben muß. Dieses ist insbesondere dann nötig, wenn der Puffer zum ersten Mal bzw. unter einem alternativen Namen gesichert werden soll.

2.2.2.2 Entwicklungsumgebung

Nachdem im vorangegangenen Abschnitt die Grundzüge des `emacs` Editors beschrieben worden sind, sollen jetzt einige Möglichkeiten vorgestellt werden, wie `emacs` beim Programmieren eingesetzt werden kann. Hier unterscheidet sich `emacs` auch wesentlich vom `vi`, der nicht annähernd die Funktionalität aufweist.

C/C++ Modus Wie bereits eingangs angedeutet, „versteht“ `emacs` Quelltexte verschiedener Programmiersprachen. Konkret heißt das, daß Quelltexte graphisch aufbereitet in einem Fenster dargestellt werden. So werden zum Beispiel Schlüsselwörter, Variablen und Makros in jeweils anderen Farben oder Schriftarten angezeigt. Dieses erhöht die Lesbarkeit von Programmen wesentlich.
Neben der Darstellung bietet `emacs` eine erweiterte Funktionalität, die speziell für C bzw. C++ Quelltexte zugeschnitten ist. Der C++ Modus[22] wird explizit durch

ESC x `c++-mode` RETURN

aktiviert.

Ein wichtiger Faktor neben einer farblichen Darstellung zur Lesbarkeit ist die Formatierung von Quelltexten[23]. Per Kommando formatiert `emacs` C/C++ Programmtexte.

Tastenfolge	Auswirkung
ESC CTRL + \	formatiert den Bereich zwischen Marke und Cursor.
ESC m	bewegt den Cursor zum ersten Nicht–Leerzeichen in der aktuellen Zeile.
ESC ;	fügt Kommentar in der aktuellen Zeile an.
ESC j	fügt Kommentar in der nächsten Zeile an.
CTRL + c CTRL + q	rückt eine Funktions– oder Klassendefinition ein.

[22] Der C Modus wird entsprechend aktiviert.

[23] Es sollte klar sein, daß die verschiedenen Farben und Schriftarten temporär, eine Formatierung jedoch permanent ist.

ESC CTRL + q	rückt einen geklammerten Ausdruck ein. Der Cursor muß sich auf der geöffneten Klammer befinden.

Besonders hilfreich sind die beiden Kommandos zum Einfügen von Kommentaren. Durch die Eingabe von ESC ; wird in eine bestimmte Spalte[24] der aktuellen Zeile gesprungen und eine Kommentarklammer `//`[25] angehängt. Anschließend kann der Kommentar eingegeben werden. Wird während dieser Eingabe der ESC ; Befehl eingeben, so wird in der nachfolgenden Zeile der Kommentar nebst Klammerung fortgeführt.

Die nachfolgende Tabelle listet weitere Befehle im C++ Modus auf.

Tastenfolge	Auswirkung
ESC CTRL + a	springt zum Beginn des Funktionsrumpfes.
ESC CTRL + e	springt zum Ende der Funktion.
ESC CTRL + h	markiert das Ende der Funktion und springt zum Anfang.
CTRL + c CTRL + v	gibt die aktuelle Version des C++ Modus aus.
CTRL + c CTRL + e	expandiert die Makrodefinition im markierten Bereich.
CTRL + c CTRL + c	kommentiert markierten Bereich ein.

[24] Die entsprechene Variable ist `comment-column`.

[25] Im C Modus wird mit `/*` und `*/` Kommentar geklammert.

etags Ähnlich wie `ctags` (vgl. Abschnitt 2.2.1.2) arbeitet das `etags` Werkzeug. `etags` ist im wesentlichen ein Suchprogramm, welches in auf mehreren Dateien verteilten Quelltexten nach bestimmten Begriffen (*tags*) sucht.

Tastenfolge	Auswirkung
ESC .	sucht nach einem Begriff.
ESC ,	sucht nach weiteren Vorkommen des gewünschten Begriffs.

Mit ESC . wird der `emacs` Editor aufgefordert, nach einem Begriff zu suchen. Standardmäßig wird nach dem Wort gesucht, welches an der aktuellen Cursorposition sich befindet. Ist der Begriff gefunden, wird ein Fenster und ein Puffer mit der entsprechenden Datei geöffnet und der Cursor positioniert. Weitere Vorkommen können mit ESC , untersucht werden. Bevor die Suche gestartet werden kann, muß natürlich `etags` gestartet werden. Dieses geschieht entweder von einer UNIX™ Shell oder mit dem ESC ! Befehl, der es erlaubt, aus dem Minipuffer heraus Shell–Kommandos abzusetzen. Die Aufrufoptionen sind im wesentlichen äquivalent zum `ctags` (vgl. Abschnitt 2.2.1.2) Werkzeug für den `vi` Editor.

make `emacs` unterstützt die Programmiererin bzw. den Programmierer nicht nur bei der

Quelltexteingabe, sondern hilft ihr bzw. ihm auch beim Übersetzen und bei der Fehlersuche (siehe nächsten Abschnitt).

`make` ist ein weiteres Werkzeug, welches dafür sorgt, daß die richtigen Quelltexte übersetzt und gebunden werden. In Abschnitt 2.3 wird darauf im einzelnen eingegangen. An dieser Stelle sei nur angemerkt, daß i.allg. `make` aufgerufen wird und der Compiler direkt.
Ist der Quelltext eingegeben und soll übersetzt und gebunden werden, d.h. es ist das `make` Werkzeug aufzurufen, so ist dieses mit dem

[ESC] [x] `compile` [RETURN]

Kommando möglich. Anschließend ist im Minipuffer der Übersetzungsbefehl, standardmäßig `make -k`[26] einzugeben und mit [RETURN] abzuschließen. `emacs` öffnet einen Puffer `*compilation*` und ein Fenster, in dem die Ausgaben aus der `make` Ausführung angezeigt werden.
Interessant ist der Fall, wenn während des Übersetzungsvorganges ein oder mehrere Fehler auftreten. Es ist nun möglich, mit dem Kommando[27]

[CTRL] + [x] [`]

an die Stelle im Quelltext zu springen, an der der Fehler aufgetreten ist. Nachfolgende [CTRL] + [x] [`] Befehlseingaben springen sukzessive zu den weiteren Fehlern. Hierdurch wird es erleichtert, Fehler an den richtigen Stellen im Quelltext zu korrigieren.

Debugger Modus Ist der Übersetzungsvorgang erst einmal erfolgreich verlaufen und ein ausführbares Programm gebunden worden, so ist i.allg. der Entwicklungsprozeß noch nicht abgeschlossen, sondern es beginnt das Testen und die Fehlersuche. Debugger helfen der Entwicklerin bzw. dem Entwickler, Fehler zu suchen. Diese werden ausführlich in den Abschnitten 3.3.4 und 3.3.5 besprochen. `emacs` besitzt eine Schnittstelle zu den Debugger `gdb` und `dbx` . Um den Debugger aus dem `emacs` Editor zu starten, ist mit

[ESC] [x] `gdb` [RETURN]

bzw. mit

[ESC] [x] `dbx` [RETURN]

in den entsprechenden Modus zu wechseln. Die Befehle, die die Debugger bieten, sind in den genannten Abschnitten aufgeführt.

`emacs` ist ein Editor, der eine ungeheure Funktionalität bietet, die in vollem Umfang kaum jemand in Anspruch nehmen wird. Für weitere Informationen sei auf die angegebene Literatur verwiesen. Wie bereits anfangs kurz dargestellt, existierte eine weitere `emacs` Variante, nämlich der `xemacs`[28]. In [Mün95] wird eine Einführung gegeben. Insbesondere sei auf das Zusammenwirken mit `oobr` bzw. `xoobr` verwiesen, welches ermöglicht, komfortabel C++ Klassen, Funktionen etc. zu verwalten.

Aufgabe 2.2
Informieren Sie sich über die Arbeitsweise des `emacs` Editors. Einen guten Einstieg bietet das Tutorial, welches mit [CTRL] + [h] [t] gestartet wird.

- **a** Wie lauten die Kommandos, um aus dem Editor heraus zu drucken ?
- **b** Wie werden Abkürzungen definiert und eingesetzt ?
- **c** Wie lassen sich wiederkehrende Aktionen als Makro definieren ?

[26] Dieser Vorgabewert kann mit der Variable `compile-command` verändert werden.
[27] Es handelt sich hierbei um ein Rückwärtsapostroph.
[28] Version 19.13

2.2.3 Weitere Editoren

In diesem Abschnitt werden kurz einige weitere Editoren vorgestellt, die z.T. Erweiterungen der beiden besprochenen Editoren `vi` und `emacs` sind.

2.2.3.1 XView `textedit`

Der `textedit` der graphischen Bedienoberfläche XView ist ein einfacher Editor, der nur die Grundfunktionen einer Textverarbeitung bietet. Hingegen ist die Bedienung intuitiv zu erlernen. Eine Funktion, die beim Programmieren nützlich ist, befindet sich unter dem Menüpunkt *Find Marked Text*. Diese erlaubt es, bestimmte Klammerungen einzufügen und zu suchen. So ist es zum Beispiel möglich, einen Textbereich zu markieren und diesen mit den Kommentarklammern `/*` und `*/` zu versehen.

2.2.3.2 `vim` und `elvis`

Der `vi` Editor ist auf viele Plattformen portiert und oftmals um einige interessante Erweiterungen ergänzt worden.
Der `vim` („vi improved“) bietet zum Beispiel die Möglichkeit, eine undo–Operation wieder rückgängig zu machen („redo“). Ebenfalls ist es möglich, im Eingabemodus die Cursor–Tasten zu benutzen, d.h. die Betätigung einer der Tasten wird als Kommando und nicht als Text interpretiert.

Wie der `vim` ist der `elvis` eine Erweiterung des ursprünglichen `ex` bzw. `vi`–Editors. Auf eine ausführliche Beschreibung sei auf das Buch [HHMK94] verwiesen.
Der `elvis` besitzt eine Reihe weiterer interner Optionen, die mit dem weiter oben beschriebenen `set` Kommando verändert werden können. Insbesondere sei auf die beiden zusätzlichen `ex`–Kommandos hingewiesen, die es erlauben, aus dem Editor heraus den Compiler oder das `make`–Werkzeug aufzurufen.

Parameter	Voreins.	Bedeutung
`cc`	`cc`	gibt den Namen und die Parameter des C/C++ Compilers an, der durch den $\boxed{:}$ $\boxed{c}$ $\boxed{c}$ Befehl aufgerufen wird. Der Wert muß in Hochkommata eingeschlossen angegeben werden.
`make`	`make`	gibt den Namen und die Parameter des `make` Werkzeuges an, welches durch den $\boxed{:}$ $\boxed{m}$ $\boxed{a}$ $\boxed{k}$ $\boxed{e}$ Befehl ausgeführt werden soll. Der Wert muß in Hochkommata eingeschlossen angegeben werden.

Wird mit einem der beiden genannten Kommandos der Compiler oder `make` gestartet, so wird die Fehlerausgabe des aufgerufenen Programmes in ein Datei umgeleitet. `elvis` durchsucht diese Datei und springt in die Zeile des ersten gemeldeten Fehlers. Nun kann die Programmiererin oder der Programmierer den Fehler korrigieren.
Durch die Eingabe von $\boxed{*}$ wird dann der Cursor an den Anfang der Zeile des nächsten Fehlers gewechselt.

An dieser Stelle soll noch auf ein Werkzeug hingewiesen werden, welches ähnlich wie `ctags` das Auffinden von Suchbegriffen in verteilten Quelltexten vereinfacht.
`ref` ist ein Werkzeug zum Auffinden und Anzeigen von Prozedurköpfen. Wird es mit einem Funktionsnamen als Parameter aufgerufen,

so wird die `tags` Datei[29] durchsucht und der Dateiname und der gesuchte Prozedurkopf mit eventuellen zusätzlichen Informationen (Kommentare oder Parameter) angezeigt [30].
`ref` arbeitet mit `elvis` zusammen. Wenn der Cursor auf einem Funktions– oder Prozedurnamen positioniert wird, kann mit dem `K`–Kommando automatisch `ref` ausgeführt werden. Das Ergebnis der Suche wird im Editor angezeigt.

Anhand des folgenden Beispiels[31] soll abschließend gezeigt werden, wie eine kontextsensitive Hilfefunktion als Makro definiert werden kann. Das Beispiel faßt einige der wesentlichen Kommandos zusammen.
Die grundlegende Idee ist es, vor dem Wort, auf welchem sich der Cursor zum Zeitpunkt der Betätigung der Funktionstaste F11 (im Kommandomodus !) befindet, die Zeichenkette `!man 3` einzufügen. Diese Zeichenkette wird zusammen mit dem nachfolgenden Schlüsselwort zuerst in den Puffer z kopiert und dieser Inhalt dann als Kommando im `ex`–Modus interpretiert.
Das Shell–Kommando `man 3` führt eine Suche im dritten Abschnitt des Online–Handbuchs nach dem angegeben Schlüsselwort durch und zeigt den Inhalt auf dem Bildschirm an.
Ein Eintrag in der `.exrc`–Datei[32] könnte wie folgt aussehen:

```
map ^[[23~ :ma z^Mwbi!man 3 ^[6h"z
y3Wd2W:@z^M:'z^M
```

Die ausgezeichneten Tasten werden durch eine Escape–Sequenz und weiterer Zeichen dargestellt. Die Zeichenfolge der Funktionstasten ist systemabhängig.

Tastenfolge	Beschreibung
`:` `m` `a` `p` `␣`	Mit diesem `ex`–Befehl wird eine Makrodefinition eingeleitet.
`CTRL` + `v` `F 11` `␣`	Die Funktionstaste F11 wird mit dem Makro belegt.
`:` `m` `a` `␣` `z`	Markiert die aktuelle Position des Cursors mit der Marke z.
`CTRL` + `v` `RETURN`	Der vorherige `ex`–Befehl muß mit einem Neue–Zeile Zeichen abgeschlossen werden.
`w` `b`	Bewegt den Cursor auf das erste Zeichen des momentanen Wortes.
`i` `!` `m` `a` `n` `3`	Geht in den Einfügemodus und schreibt die Zeichenkette `!man 3` in den aktuellen Text.
`CTRL` + `v` `ESC`	Beendet den Einfügemodus.
`6` `h`	Bewegt den Cursor sechs Zeichen zurück nach links.
`"` `z` `y` `3` `W`	Kopiert die nächsten drei Wörter in den Puffer z.

[29] Standardmäßig wird das aktuelle Verzeichnis untersucht. In der Umgebungsvariable `TAGPATH` können weitere durch Doppelpunkt getrennte Verzeichnisse angegeben werden, die zusätzlich nach `tags` Dateien durchsucht werden.

[30] Die `ctags` Version, die zusammen mit `elvis` und `ref` verbreitet wird, versteht auch C++ Quelltexte, wenn mit der Schaltereinstellung `-C -d -t -T` aufgerufen wird. Hierdurch werden zusätzlich Klassenmethoden, C–Präprozessor Anweisungen `#define` und weitere mögliche Suchbegriffe in die `tags`–Datei aufgenommen.

[31] Das Beispiel bezieht sich auf den `elvis`–Editors. Eine Anpassung an andere `vi` Implementierungen ist eventuell möglich. Solch ein relativ aufwendiges Makros ist jedoch auf dem von SUN ausgelieferten Editor `vi` nicht möglich, da ein Ausschneiden von Zeichenketten mit dem `y`–Befehl innerhalb eines Makros nicht unterstützt wird.

[32] Die Makrodefinition muß in **einer** Zeile stehen.

d 2 W	Löscht die zwei nächsten Wörter wieder aus dem Text heraus.
: @ z	Führt die im Puffer z stehende Zeichenkette als Befehl aus.
CTRL + v RETURN	Der ex–Befehl muß durch ein Neue–Zeile Zeichen abgeschlossen werden.
: ' z	Bewegt den Cursor zurück an die markierte Stelle z.
CTRL + v RETURN	Schließt auch diesen letzten ex–Befehl mit einem Neue–Zeile Zeichen ab.

2.2.3.3 `jed` und `jove`

Die beiden Editoren `jed` und `jove` sind weitere Versionen, die den oben beschriebenen `emacs` implementieren. An dieser Stelle sei auf die ausführlich vorhandene Literatur verwiesen.

2.3 Der Kommandogenerator `make`

Gewöhnlich befinden sich die Quelltexte für das zu entwickelnde Programm in mehreren Dateien verteilt. Dieses dient der Modularisierung und hebt die Übersicht an. Fernerhin wird oftmals auch auf vorhandene Quelltexte zurückgegriffen.
In C/C++ werden meistens Deklarationen in den sogenannten **Header–Dateien**, die als **Suffix** `.h` besitzen[33], aufgeführt. Die eigentlichen Definitionen befindet sich dann in Dateien mit den entsprechenden Suffixen `.c` für C– bzw. `.cc` für C++ –Quelltexte[34].
Um aus den Quelltexten ein ausführbares Programm zu erhalten, müssen die Dateien zuerst übersetzt und dann anschließend gebunden werden. Probleme treten immer dann auf, wenn einzelne Dateien verändert und infolgedessen neu übersetzt werden müssen. Hier ergibt sich oftmals schnell ein inkonsistenter Zustand, da nicht alle notwendigen Dateien kompiliert worden sind. Prinzipiell ist es natürlich möglich, alles neu zu übersetzen und zu binden. Der damit verbundene Zeitaufwand ist in den meisten Fällen jedoch nicht nötig.
Der **Kommandogenerator** `make` von **Stuart Feldman**, eines der effizientesten Werkzeuge zur Verwaltung von Softwareprojekten, nimmt dem Entwickler bzw. der Entwicklerin diese Aufgaben ab. Das mögliche Anwendungsgebiet von `make` läßt sich sehr weit fassen; an dieser Stelle soll jedoch nur eine kurze Einführung in den Bereich der Softwareentwicklung gegeben werden (vgl. [Fel79] und [OT91]).
Die Arbeitsweise bereitet der Anfängerin bzw. dem Anfänger oftmals Schwierigkeiten, da hier kein strikt sequentieller Ablauf durchgeführt wird (die benutzte Technik erinnert ein wenig an die deklarative Programmiersprache **Prolog**).

2.3.1 Einführung

Beim Aufruf von `make` wird eine Datei `Makefile` bzw. `makefile` eingelesen und die darin enthaltenen Anweisungen ausgeführt. Der generelle Eintrag hat folgendes Format:

Ziel ... **:** *[Abhängigkeit ...]*
[Kommando]
...

Ein solcher Eintrag besteht im wesentlichen aus drei Teilen:

[33] Die Namesgebung ist natürlich nicht zwingend erforderlich. Man erspart sich aber viele Unannehmlichkeiten, wenn diese Konventionen beibehalten werden.

[34] In **Modula–2** wird eine ähnliche, wenn auch striktere, Zerlegung in Definitionsmodul (Exportschnittstelle) und Implementierungsmodul (eigentlicher Inhalt) angesetzt.

- Unter *Ziel* sind die Dateien oder Ausführungsschritte aufgeführt, die erzeugt bzw. durchgeführt werden sollen.
- Unter *Abhängigkeit* sind die Bedingungen (Dateien oder Ausführungsschritte) aufgeführt, von denen das Ziel abhängt.
- Die nachfolgenden Zeilen *Kommando* enthalten die Anweisungen, die ausgeführt werden müssen, um das Ziel zu erreichen. Dieser Bereich wird auch als **Regel** bezeichnet. Den Anweisungen muß jeweils ein Tabulatorzeichen (keine Leerzeichen) vorangestellt sein.
  Bei der Abarbeitung der Regel wird für jede Kommandozeile eine eigene **Bourne shell** aufgerufen, in der dann die Befehle ausgeführt werden[35]. Sollen mehrere Befehle in einer shell ablaufen, dann müssen diese durch ein Semikolon getrennt in einer Zeile aufgeführt werden.

Aus den Zielen und Abhängigkeiten läßt sich ein Abhängigkeitsgraph aufstellen. Beim Abruf von `make` wird dieser Graph in Form einer Tiefensuche, also zuerst die Teilgraphen und dann die Wurzel, durchlaufen. Handelt es sich bei den Knoten um Dateien (im momentanen Verzeichnis), werden deren Modifikationszeiten verglichen. Ist die Wurzel älter als mindestens einer der Kinderknoten, wird die Regel ausgeführt. Handelt es sich nicht um existierende Dateien bei den Knoten, dann wird der aktuelle Zeitpunkt zum Vergleich benutzt.

Die Ziele und Abhängigkeiten können also Dateien oder allgemeine Ausführungsschritte (dummy targets bzw. dummy prerequisites) sein. Hieraus ergeben sich folgende Konsequenzen:

- Ein Ausführungsschritt vor dem Doppelpunkt (dummy target) ist immer älter als deren Abhängigkeiten, d.h., die entsprechende Regel wird auf jeden Fall ausgeführt.
- Ein Ausführungsschritt nach dem Doppelpunkt (dummy prerequisite) ist immer neuer als die aufgeführten Ziele, d.h., die Regel des Ziels wird auf jeden Fall ausgeführt.

Folgendes Beispiel soll den Zusammenhang verdeutlichen.

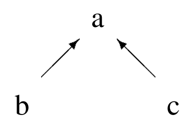
```
# Makefile

a :     b c
        touch a

b :
        touch b

c :
        touch c
```

Kommentare werden durch ein # Zeichen eingeleitet und durch ein Zeilenende–Zeichen beendet. Endet eine Zeile mit dem \–Zeichen, wird in der darauffolgenden Zeile fortgefahren. Leerzeilen werden überlesen.



Der Abhängigkeitsgraph läßt sich wie oben darstellen. Die Arbeitsweise vom `make`–Programm läßt sich wie folgt beschreiben:

1. Das Ziel `a` hängt von `b` und `c` ab. Die Ziele `b` und `c` besitzen keine Abhängigkeiten.
2. Es existiert keine Datei `b`, und da keine weiteren Abhängigkeiten bestehen, wird die Regel `touch b`[36] ausgeführt.

[35] Da alle Befehle der Bourne shell zur Verfügung stehen, ist das Anwendungsgebiet von `make` nicht auf den Bereich der Programmentwicklung eingeschränkt.

[36] Der `touch`–Befehl setzt die Zugriffs– und Änderungszeiten der angegebenen Datei auf die aktuelle Zeit. Wenn die Datei nicht existiert, wird sie erzeugt.

3. Es existiert keine Datei `c`, und da keine weiteren Abhängigkeiten bestehen, wird die Regel `touch c` ausgeführt.
4. Da die Abhängigkeiten von `a` neu erzeugt worden sind, wird auch die Regel `touch a` für `a` ausgeführt.

Ist kein Ziel als Parameter beim Aufruf von `make` angegeben, so wird des erste Ziel (hier also `a`) als Wurzel des Abhängigkeitsbaumes benutzt. Durch z.B. den Aufruf von `make b` wird der Teilbaum beginnend beim Knoten `b` abgearbeitet.
Durch diesen Baumdurchlauf wird auch schnell deutlich, daß die Reihenfolge der Einträge in der Datei `makefile` nicht entscheidend ist. Dieser Umstand verursacht bei der Anfängerin und beim Anfänger erfahrungsgemäß Schwierigkeiten.

Das folgende Beispiel zeigt, wie der Kommandogenerator bei der Entwicklung von Software benutzt werden kann. Die Ausgangslage sei gegeben durch die Dateien `x.c`, `y.c`, den zugehörigen Header–Dateien und `main.c`. In `main.c` wird auf Deklarationen aus `x.h` und `y.h` zurückgegriffen. Das `Makefile`[37] für das fiktive Programm `z` könnte dementsprechend folgende Gestalt haben:

```
# Makefile fuer Programm z
z :      x.o y.o main.o
         cc -o z x.o y.o main.o

x.o :    x.c
         cc -c x.c

x.c :    x.h
         touch x.c

y.o :    y.c
         cc -c y.c

y.c :    y.h
         touch y.c

main.o : main.c x.h y.h
         cc -c main.c
```

[37] Die einzelnen Befehle werden erst in Abschnitt 2.5 näher erläutert. An dieser Stelle sei vermerkt, daß `cc -c x.c` eine Datei `x.c` in eine Objektdatei `x.o` übersetzt. `cc -o z x.o y.o main.o` bindet die angegebenen Objektdateien zu einem ausführbaren Programm `z` zusammen.

Der zugehörige Abhängigkeitsgraph ist kein Baum; die Vorgehensweise aus dem ersten Beispiel ist jedoch übertragbar.

Oftmals wird ein Ausführungsschritt `clean` als Ziel angegeben; per Konvention werden bei der Ausführung sämtliche Objektdateien etc. gelöscht. Hierdurch wird eine komplette Neuübersetzung möglich.

```
clean :
        rm -f *.o z core
```

Das letzte Beispiel zeigt, daß einige Einträge sich recht ähnlich sind. Der nächste Abschnitt soll zeigen, wie solche Ähnlichkeiten zusammengefaßt werden können.

Aufgabe 2.3
Machen Sie sich mit dem Kommandogenerator `make` vertraut.

a Worin liegt der Unterschied zwischen den folgenden zwei Regeln zu den Zielen `z1` und `z2` ?

```
z1 :
        cd /tmp
        pwd

z2 :
        cd /tmp ; pwd
```

b Welche Bedeutung haben die Regeln zu den Zielen `x.c` und `y.c` ?

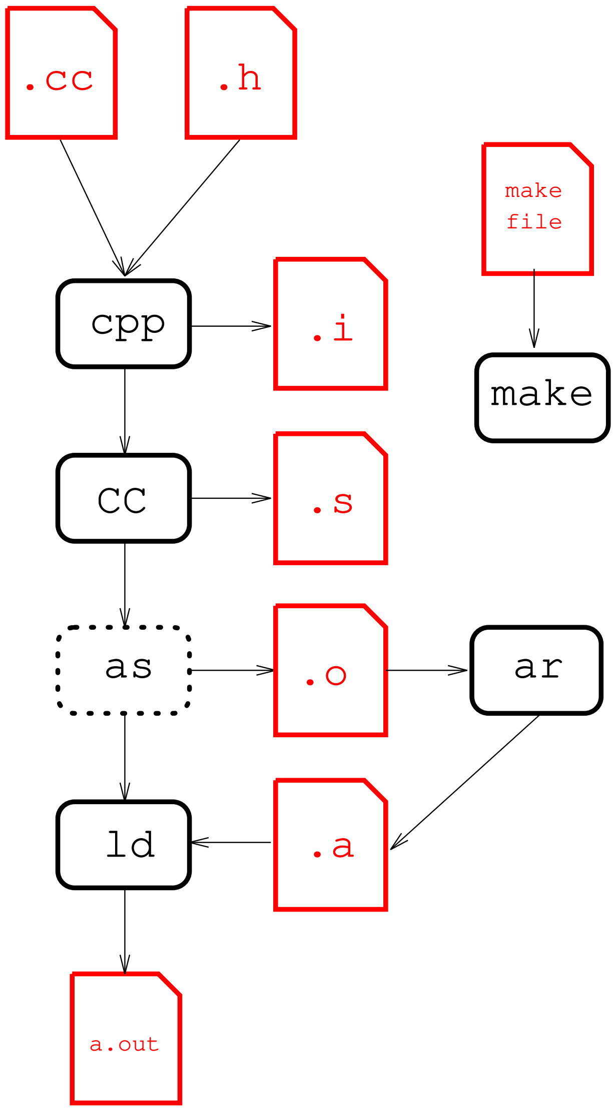


Abbildung 2.5: Ablauf der Programmerstellung

2.3.2 Makros und Suffix–Regeln

Die Benutzung von **Makros** reduziert die Schreibarbeit. Hierdurch wird die Lesbarkeit erhöht. Im wesentlichen handelt es sich hierbei um textuelle Ersetzungen, d.h. Zeichenketten werden durch (andere) Zeichenketten automatisch vom `make`–Werkzeug substituiert.
Makros werden durch

```
NAME = Zeichenkette
```

definiert[38]. Um auf definierte Makros zu verweisen, müssen diese mit `${NAME}` oder `$(NAME)` angegeben werden. Besteht der Name nur aus einem Buchstaben, so kann auf die Klammern verzichtet werden.
Fernerhin ist es erlaubt, Makros in Definitionen anderer zu benutzen, z.B. also

```
SUFFIX = tar
DATEI  = xyz.${SUFFIX}
```

Des weiteren generiert `make` beim Bearbeiten einer jeden Abhängigkeitszeile automatisch einige Makros, die in der nachfolgenden Regel benutzt werden können.

`$?` Die Liste der Abhängigkeiten, also Dateien oder Ausführungsschritte (dummy prerequisite), die jünger als das aktuelle Ziel sind.

`$@` Der Name des aktuelles Ziels.

Das Makro `$$@` entspricht `$@` mit dem Unterschied, daß das erstgenannte nur in der Abhängigkeitszeile (rechts vom Doppelpunkt) benutzt werden darf.

Das Beispiel aus dem letzten Abschnitt kann also wie folgt vereinfacht werden:

[38] Üblicherweise werden Makros in Großbuchstaben angegeben; wie jedoch bereits an anderer Stelle erwähnt, ist dieses auch nur Konvention. Wird diese beibehalten, ist es für andere leichter, die Zusammenhänge im `Makefile` zu verstehen.

```
# Makefile mit Makros
# fuer Programm z
OBJECTS = x.o y.o main.o
CC      = cc   # gcc fuer GNU cc
CFLAGS  =      # -g zum Debuggen

z :     ${OBJECTS}
        ${CC} -o $@ ${OBJECTS}

x.o :   x.c
        ${CC} ${CFLAGS} -c $?

x.c :   x.h
        touch x.c

y.o :   y.c
        ${CC} ${CFLAGS} -c $?

y.c :   y.h
        touch y.c

main.o : main.c x.h y.h
        ${CC} ${CFLAGS} \
        -c main.c
```

Es werden einige zusätzliche Makros `${CC}` und `${CFLAGS}` definiert, die zwar auf dem ersten Blick zusätzlichen Aufwand bedeuten: jedoch lassen sich Veränderungen des Übersetzers und der Optionen dadurch einfach bewerkstelligen.

Eine weitere Möglichkeit zur Vereinfachung und Reduzierung ergibt sich durch die Einführung von sogenannten **Suffix–Regeln**.
Das bisherige Beispiel läßt sich unter Verwendung der Suffix–Regeln weiter vereinfachen.

```
# Makefile mit Makros und
# Suffix-Regeln fuer Programm z
OBJECTS = x.o y.o main.o
CC      = cc   # gcc fuer GNU cc
CFLAGS  =      # -g zum Debuggen

.SUFFIXES :     .o .c .h

.c.o :
          ${CC} ${CFLAGS} -c $<

.h.c :
```

```
            touch $@

z :         ${OBJECTS}
            ${CC} -o $@ ${OBJECTS}

main.o :    main.c x.h y.h
            ${CC} ${CFLAGS} -c \
            main.c
```

Das Makro `$<` hat dieselbe Bedeutung wie `$?` mit dem Unterschied, daß das erstgenannte Makro nur in Suffix–Regeln benutzt werden darf.
Als erstes fällt die `.SUFFIXES` Zeile auf. Hierdurch wird erklärt, daß eine Datei mit der Endung (Suffix) `.o`, `.c` oder `.h` besonders zu behandeln ist.
Ist nämlich keine Abhängigkeit explizit definiert, so wird eine entsprechende Suffix–Regel angewendet. Diese haben folgendes Format

.Suffix1.Suffix2 :
[Kommando]
...

und wird gelesen als: Eine Datei `file.`*Suffix2* (Ziel) ist abhängig von der Datei `file.`*Suffix1* (Abhängigkeit). Ist die Abhängigkeit jünger als das Ziel, so wird die angegebene Regel ausgeführt.
Im aufgeführten Beispiel werden folgende Schritte bearbeitet:

1. Das Ziel `z` ist abhängig von `${OBJECTS}`.
2. Die Datei `x.o` existiert nicht. Daher wird überprüft, von welchen Bedingungen diese ihrerseits abhängig ist. Es existiert keine explizit definierte Abhängigkeit, weshalb die Suffix–Regel `.c.o` angewendet und `x.c` zu `x.o` neu übersetzt wird. Vorher wird noch überprüft, ob die Kommandos der Suffix–Regel `.c.h` angewendet werden müssen.
3. Dasselbe wird für die Datei `y.o` bzw. `y.c` durchgeführt.
4. `main.o` existiert ebenfalls nicht, allerdings gibt es eine explizit definierte Abhängigkeit. Deswegen wird diese bearbeitet und nicht die Suffix–Regel `.c.o` .
5. Alle Abhängigkeiten sind überprüft worden. Da diese jünger als das Ziel `z` sind, wird die Regel ausgeführt und `z` neu gebunden.

Diese Beispiele zeigen nur einen kleinen Ausschnitt aus den Möglichkeiten, die das `make`–Werkzeug bietet.

Abschließend werden einige Schalter, die als Parameter beim Aufruf übergeben werden können, kurz beschrieben.

`make` *[*`-f` *makefile] [Optionen] [Ziele] [Makrodefinitionen]*

Mögliche Optionen sind:

`-d` Debug–Modus erzeugt Bildschirmausgaben zur Kontrolle.

`-f` Die angegebene Datei *makefile* wird anstatt `Makefile` oder `makefile` bearbeitet.

`-n` Die Kommandozeilen werden zwar auf dem Bildschirm ausgegeben, die Anweisungen jedoch nicht ausgeführt.

`-s` Es werden keine Ausgaben bei der Abarbeitung auf dem Bildschirm gedruckt (silent).

2.3.3 Weitere Werkzeuge

`make` ist ursprünglich für das UNIX™ –Betriebssystem entwickelt worden, steht inzwischen jedoch auch auf andere Plattformen (MS-DOS™, OS/2™) zur Verfügung.
Das `make`–Werkzeug ist mehrfach verbessert und erweitert worden (z.B. `mk` und `nmake`). Insbesondere existiert auch eine Version der Free Software Foundation (GNU).
`imake` ist eine Art Meta–make, welches die Portierung von Software auf verschiedene

(UNIX™) Plattformen unterstützt.
makedepend erlaubt es, die Abhängigkeiten der eingebundenen Header–Dateien (siehe Abschnitt 2.4.2) zu erkennen.
Die Programme sccs (source code control system) und rcs (revision control system) dienen zur Verwaltung von Versionen von Quellcodedateien. Dieses ist besonders wichtig, wenn verschiedene Personen (Team) gleichzeitig in einem Softwareprojekt implementieren und daher Änderungen konsistent verwaltet werden müssen. shape vereinigt make und sccs.

Fernerhin existieren auch parallele Versionen von make, so z.B. pmake aus dem Amoeba–Projekt (verteiltes Betriebssystem) und Lmake der TU Braunschweig (vgl. [LS94, Seite 287]).

2.4 Der Präprozessor cpp

Bevor der Quelltext vom eigentlich Übersetzer (s.u.) bearbeitet wird, durchläuft dieser den Präprozessor cpp, der verschiedene Aufgaben durchführt. Hierzu gehören u.a. folgende Bearbeitungsschritte:

- Zeilen, die mit einem \ enden, werden mit dem Text der nächsten Zeile unter Auslassung des \ und Neue Zeile–Zeichens zusammengefügt.
- Aufeinanderfolgende Zeichenketten–**Literale** (in gewisser Weise Konstanten) werden zusammengefügt, z.B.

  ```
  cout <<
   "Universitaet" " Hildesheim";
  ```

  wird in die folgende Anweisung übersetzt

  ```
  cout <<
   "Universitaet Hildesheim";
  ```

- Präprozessoranweisungen, im folgenden als **Direktive** bezeichnet, werden ausgeführt.

Der letzte Punkt soll etwas ausführlicher besprochen werden.
Direktiven werden durch ein #–Zeichen in der ersten Spalte (ganz links) einer Zeile eingeleitet.

2.4.1 Makros

Makros stellen einen Mechanismus zum Ersetzen von Symbolen zur Verfügung. Ein einfaches Makro ist z.B.

```
#define DAYS_OF_MONTH 31
```

Der Präprozessor ersetzt alle Vorkommen von DAYS_OF_MONTH durch 31. Dementsprechend aus

```
int december[DAYS_OF_MONTH];
```

wird

```
int december[31];
```

Ähnlich wie in anderen prozeduralen Sprachen ist es möglich, Funktionen zu definieren. Ein solches **funktionales Makro** könnte sein:

```
#define SQUARE(X) ((X)*(X))
```

Allerdings wird hier nur eine textuelle Ersetzung durchgeführt. So wird aus

```
int x=7;
int y=SQUARE(x):
```

folgender Quelltext

```
int x=7;
int y=((x)*(x));
```

Generell läßt sich sagen, daß Makros nur mit Bedacht eingesetzt werden sollten. Viele Möglichkeiten lassen sich durch vorhandene Konstrukte ausdrücken (Ausnahmen siehe Abschnitt 3.3.3 und 3.4.1). Konstanten werden in C++ durch das Schlüsselwort const beschrieben.

```
const int DAYS_OF_MONTH=31;
```

Der Vorteil ist, daß hier eine Typenkontrolle stattfindet. Auch lassen sich die funktionalen Makros weitestgehendst vermeiden.

2.4.2 Einbinden von Quelltextdateien

Aus verschiedenen Gründen, z.B. zur Einteilung in Deklarations– und Definitionsteil, komponentenweise Übersetzung oder einfach nur zur Übersichtlichkeit, wird der Quelltext i.allg. auf mehrere Dateien aufgeteilt.
Um auf Bezeichner in anderen Dateien zugreifen zu können, müssen diese beim Übersetzungsvorgang eingelesen werden. Dieses geschieht durch die folgende Direktive:

```
#include <NAME>
```

oder durch

```
#include "NAME"       ,
```

wobei `NAME` durch den Dateinamen ersetzt werden muß.
Wird die erste Variante benutzt, wird in den voreingestellten Verzeichnissen nach der spezifizierten Datei gesucht. Bei der zweiten Möglichkeit wird hingegen zunächst im aktuellen Verzeichnis nach der betreffenden Datei gesucht.
Allgemein gilt die Konvention, daß Systemdateien mit `<NAME>` dazugeladen werden, während bei benutzerdefinierten Dateien die letztere Variante vorzuziehen ist.

2.4.3 Bedingte Übersetzung

Mit weiteren Direktiven ist es möglich, bestimmte Quelltextzeilen von der Übersetzung durch den eigentlichen C/C++ Übersetzer auszuschließen. Dieses kann insbesondere dann notwendig sein, wenn portabler, d.h. auf verschiedenen (Betriebs–) Systemen einsetzbarer, Quelltext entwickelt werden soll.

```
#ifdef _MSDOS_
#include <conio.h>
#define CLRSCR clrscr()
#else
#include <stdlib.h>
#define CLRSCR system("clear")
#endif
```

In diesem Beispiel wird abhängig von der Definition des Symbols `_MSDOS_` ein Makro `CLRSCR` definiert, welches erlaubt, den Bildschirm auf MS-DOS™– bzw. UNIX™–Systemen zu löschen.

```
main() {
 CLRSCR;
 cout << "ok." << endl;
}
```

Aufgabe 2.4
Informieren Sie sich über die Syntax und die Verhaltensweise des Präprozessors `cpp`.

a Welche Bedeutung haben die `#`– und `##`–Operatoren ?

b Worin liegt der Unterschied zwischen

```
int square(int x) {
 return x*x;
}
```

und dem Makro

```
#define SQUARE(X) (X*X)
```

c Sogenannte **Headerdateien** sind i.allg. nach folgendem Muster aufgebaut:

```
#ifndef _NAME_OF_HEADERFILE
#define _NAME_OF_HEADERFILE

/* hier folgt jetzt der
   C/C++ Quelltext       */

#endif
```

Erklären Sie den Sinn des beschriebenen Schemas. Was passiert, wenn zwei Dateien beim Übersetzungsvorgang eine Datei einbinden, die nicht diesem Schema folgt ?

d Welche Bedeutung haben die Makros `__LINE__` und `__FILE__` ?

2.5 Die Übersetzer `cc/CC` und `gcc/g++`

Die primäre Aufgabe des Übersetzers ist es, Quelltextdateien auf ihre Syntax zu überprüfen und diese in Objektdateien zu übersetzen. Im folgenden wird kurz auf die C–Übersetzer `cc` von SUN und das GNU–Äquivalent `gcc` bzw. auf die C++ Übersetzer `CC` und `g++` eingegangen.

Die Compiler leisten mehr als das eigentliche Übersetzen, vielmehr gliedert sich der Vorgang in vier sequentielle Abschnitte:

1. Bearbeitung durch den Präprozessor
2. Eigentliche Übersetzung
3. Assemblierung
4. Binden

Durch geeignete Schaltereinstellungen beim Aufruf können bestimmte Bearbeitungsschritte einzeln aufgerufen werden (vgl. Abbildung 2.5).

Wird ein C++ –Quelltext bearbeitet, ist oftmals ein weiterer Schritt notwendig. Nachdem der Quelltext durch den Präprozessor abgearbeitet wurde, wird dieser zuerst in einen C–Quelltext übersetzt und dann an den C Kompiler weitergeleitet. Dieser Vorgang wird (implizit) durch das sogenannte `cfront` Programm durchgeführt[39].
Der Aufruf lautet für den SUN C++ Übersetzer (analog für die anderen):

`CC` *[Schalter] Dateien*

Als Schalter sind die folgenden üblich.

[39] Inzwischen existieren bereits aber auch „reine" C++ –Übersetzer, die keinen C–Zwischenquelltext erzeugen. Hierdurch ist es auch möglich, C++ spezifische Optimierungen vorzunehmen.

`-c` Für jede übergebene Datei wird eine entsprechende `.o` Datei erzeugt. Das automatische Binden wird unterdrückt.

`-D`*name[=Def]* Diese Anweisung erzeugt eine Variable für den Präprozessor und belegt diese mit dem übergebenen Wert (wird sonst standardmäßig mit 1 belegt). Dieses entspricht der Anweisung `#define` *name [Def]*

`-E` Die angegebene Datei wird nur vom Präprozessor bearbeitet. Die Ausgabe erfolgt auf die Standardausgabe oder, falls mit `-o` angegeben, in eine Datei.

`-g` Es werden für die Testhilfen weitere Informationen erzeugt und in der Objektdatei abgelegt. Soll mit einer Testhilfe (z.B. dbx) gearbeitet werden, dann muß dieser Schalter angegeben sein.

`-help` Es wird eine Kurzbeschreibung auf die Standardausgabe gegeben.

`-I`*pfad* Der angegebene Pfad wird in den Suchpfad zum Auffinden der `#include`–Dateien aufgenommen.

`-l`*name* Beim Binden wird auch die angebene Programmbibliothek dazugebunden.

`-L`*pfad* Der angegebene Pfad wird in den Suchpfad zum Auffinden von Objektdateien und Bibliotheken aufgenommen.

`-o`*name* Hierdurch wird der Name der Ausgabedatei festgelegt.

`-w` Es werden keine Warnungen auf die Standardausgabe gegeben.

Der GNU Übersetzer `g++` ist ein Programm, welches den Übersetzer `gcc` mit der entsprechenden Schaltereinstellung für die Verarbeitung von C++ –Quelltextdateien aufruft.

2.6 Der Binder `ld`

Der **Binder** soll nur kurz besprochen werden, da ein expliziter Aufruf normalerweise nicht nötig ist.

`ld` erlaubt es, mehrere Objektdateien und Bibliotheken (s. nächsten Abschnitt) zu einem ausführbaren Programm zusammenzubinden. Oftmals müssen fernerhin noch bestimmte Laufzeitmodule angefügt werden.

Üblicherweise sollte der Bindevorgang vom Übersetzer `cc` oder `CC` angestartet werden, da hierdurch die notwendigen Module automatisch hinzugefügt werden. Beispiel:

```
CC -o z x.o x.o main.o
```

2.7 Der Bibliotheksverwalter `ar`

Der **Bibliotheksverwalter** `ar` ermöglicht es, Objektdateien[40] zu einer Objektbibliothek zusammenzufassen. Dieses hat u.a. folgende Vorteile:

- höhere Bindegeschwindigkeit, da der Binder für jede Bibliothek nur einmal aufgerufen wird.
- bessere Übersicht, da die einzelnen Objektdateien zusammengefaßt werden.
- ausführbare Programme werden kürzer, da nur die notwendigen Teile dazugebunden werden.
- (geringfügige) Platzersparnis, da nur eine Datei im entsprechenden Verzeichnis verwaltet werden muß.

Der Aufruf hat folgenden Aufbau:

`ar` *Funktion Bibliotheksdatei Objektdateien*

[40] Prinzipiell können von `ar` Dateien beliebigen Typs verwaltet werden.

Als *Funktion* sind die nachfolgend aufgeführten Parameter sinnvoll:

`d` Die angegebenen *Objektdateien* werden aus der *Bibliotheksdatei* gelöscht.

`m` Die genannten *Objektdateien* werden an das Ende der *Bibliotheksdatei* angefügt.

`r` Die in der *Bibliotheksdatei* aufgeführten *Objektdateien* werden ersetzt.

`t` Es wird das Inhaltsverzeichnis der *Bibliotheksdatei* ausgegeben.

Die Namen der Objektdateien, die in einer Bibliothek zusammengefaßt werden sollen, dürfen maximal 15 Zeichen lang sein; anderenfalls werden die überzähligen Zeichen ignoriert. Vorzugsweise sollten Bibliotheken als letzte Dateien beim Bindevorgang angegeben werden.
Das Programm `ranlib` mit dem Namen der Bibliotheksdatei als Paramter legt in dieser ein Inhaltsverzeichnis an. Hierdurch wird die Bindegeschwindigkeit erhöht.

Natürlich kann der Kommandogenerator `make` (siehe Abschnitt 2.3) dazu benutzt werden, um Bibliotheken immer auf dem neuesten Stand zu halten.

```
z :     main.o libmy.a
        ${CC} -o $@ main.o \
        libmy.a

libmy.a : x.o y.o
          ar r $@ $? ; \
          ranlib $@
```

Lektion 3

Fehlersuche und Testhilfen

Entsprechend der Bedeutung wird der nicht trivialen Aufgabe der **Fehlersuche** und des **Test**ens von Programm (–teilen) eine vollständige Lektion gewidmet.
Werden Fehler während der Implementierung von vornherein ausgeschlossen, dann kann diese Lektion getrost übersprungen werden.

3.1 Einführung

Fehler bei der Entwicklung lassen sich nicht vermeiden (welches etwas größeres Softwareprodukt stürzt nicht mal ab); jedoch stehen eine Menge von Werkzeugen und Methoden zur Verfügung, die einem bei der Fehlersuche während des Entwicklungsprozesses behilflich sein können.
In dieser Lektion wird ein kurzer Überblick über die Fehlersuche gegeben.

3.2 Fehlerquellen

Die möglichen Fehler, die bei der Implementierung auftreten können, lassen sich im wesentlichen den folgenden Kategorien zuteilen:

Fehler beim Übersetzen Fehler dieser Klasse lassen sich im allgemeinen einfach lokalisieren und beheben. Oftmals ist die Ursache nur ein Tippfehler (Syntaxfehler).

Fehler beim Binden Fehler dieser Klasse lassen sich hingegen schon schwieriger finden. Ursache ist in den meisten Fällen eine fehlende oder doppelte Definition. Diese Fehler treten i.allg. beim Binden von verschiedenen Modulen (Objektdateien und Bibliotheken) auf.

Laufzeitfehler Tritt ein Fehler zur Laufzeit auf, dann wird in den meisten Fällen das Programm terminiert und ein **Speicherabzug** (`core`–Datei) erzeugt. Diese Fehler können sehr unangenehm werden, insbesondere dann, wenn diese teilweise oder garnicht reproduzierbar sind.
Eine Unterklasse bilden die **mathematischen Fehler**, die im Zusammenhang mit Funktionen der mathematischen Bibliothek auftreten (z.B. Division durch null, Quadratwurzel aus einer negativen Zahl).
Zu den Laufzeitfehlern im weiteren Sinne gehören des weiteren **logische Fehler**, d.h. Implementierung und Spezifizierung stimmen nicht überein. Auf solche Fehler wird im folgenden nicht weiter eingegangen. **Korrektheit** ist ein wichtiges Kriterium bei der Bewertung von Programmen; dieses Themengebiet sollte aber an anderer Stelle diskutiert werden.

Programmieren lernt man durch Erfahrung, d.h. der Lernerfolg wird auch durch die eigenen Fehler bestimmt. Während für den Anfänger bzw. die Anfängerin die ersten beiden Fehlerklassen die ersten Hürden zum Erfolg sind, treten bei der geschulten Programmiererin bzw. beim geschulten Programmierer meistens Fehler der dritten Klasse auf.

3.3 Werkzeuge und Methoden

In den folgenden Abschnitten werden einige Möglichkeiten vorgestellt, auftretende Fehler zu untersuchen mit dem Ziel, diese dann anschließend zu beseitigen.

3.3.1 Syntaxfehler

Syntaxfehler werden durch den Übersetzer entdeckt. Dieser erzeugt beim **Übersetzungsvorgang** entsprechende Fehlermeldungen bzw. Warnungen. Der Aufbau einer Meldung gliedert sich im wesentlich in vier Abschnitte: der Name der Datei, in der der Fehler aufgetreten ist, die Zeilennummer und ob es sich um einen Fehler oder nur um eine Warnung handelt. Hieran schließt sich der eigentliche Meldungstext an.
Ein fehlerhaftes Programm

```
int main() {
 int x=5
}
```

erzeugt beim Übersetzungsvorgang folgende Fehlermeldung

```
"p1.cc", line 3: error: syntax
error Compilation failed
```

Schnell wird ersichtlich, daß das abschließende Semikolon fehlt. Leider sind nicht alle Syntaxfehler so trivial; letztendlich lassen sich diese Fehler aber schnell beseitigen. Der Übersetzer gibt schon ziemlich präzise Hinweise auf die Fehlerursache.
Oftmals erzeugen Fehler weitere Fehler (**Folgefehler**). Dementsprechend werden auch mehrere Fehlermeldungen ausgegeben. Bei der Fehlerbehebung ist es dann ratsam, zuerst den erstgemeldeten Fehler zu eliminieren und neu zu übersetzen. Die Folgefehler verschwinden dann von selbst.
Folgendes Programm erzeugt eine Meldung vom Übersetzer.

```
int main() {
 char* a,b;
 a=new char[30];
 b=a;
}
```

Der Fehlermeldungstext kann hierbei von Übersetzer zu Übersetzer variieren. Der `g++` Kompiler erzeugt folgende Meldung

```
p2.cc: In function `int main()':
p2.cc:4: warning: assignment of
integer from pointer lacks a cast
```

Hingegen berichtet der SUN-Übersetzer `CC`

```
"p2.cc", line 4: error: bad
assignment type: char  = char *
```

Bemerkenswert ist es, daß der GNU –Übersetzer nur eine Warnung ausspricht und eine lauffähige Objektdatei erstellt, während der SUN–Übersetzer einen Fehler erzeugt und den Übersetzungsvorgang abbricht.
Man ist gut beraten, auch Warnungen nicht zu ignorieren, sondern diese, wie obiges Beispiel zeigt, genauestens zu untersuchen. Es wird zwar in den meisten Fällen ein ausführbares Programm erzeugt; **Fehler zur Laufzeit** sind dann aber meistens die Folge.

Auf zwei weitere Verfahren zur Bearbeitung von schwierigen Syntaxfehlern sei kurz verwiesen. Mit dem Schalter `-E` ist es möglich, nur den Präprozessor aufzurufen. Hier bietet es sich an, die durchgeführten #–Anweisungen (z.B. Ersetzungen) zu überprüfen. Beispiel:

```
#ifndef TRUE
#define TRUE    1
#endif
#if TRUE==1
#define FALSE 0
#endif

int main () {
```

```
 int x=TRUE, y=FALSE;
}
```

`g++ -E p3.cc` erzeugt folgende Ausgabe auf den Bildschirm

```
# 1 "p3.cc"
int main () {
 int x=1 , y=0 ;
}
```

Die beabsichtigte Ersetzung der Variablen hat der Präprozessor also richtig durchgeführt. Wird hingegen mit `-DTRUE=17` übersetzt, so wird `FALSE` nicht ersetzt; folglich erzeugt der Übersetzer einen Fehler.

```
# 1 "p3.cc"
int main () {
 int x=17 , y=FALSE ;
}
```

Des weiteren steht für C–Dateien das Programm `lint` zur Verfügung, mit dem weitere umfangreichere Syntaxüberprüfungen durchgeführt werden können.

Aufgabe 3.1
Machen Sie sich mit dem Format und Inhalt der Meldungen beim Übersetzungsvorgang vertraut.

- **a** Berichtigen Sie den Fehler, der in Zeile 4 (`p2.cc`) hervorgerufen wird, so daß kein Übersetzer eine Meldung ausgibt.
- **b** Welche Funktion hat der Schalter `-D` ? Was passiert, wenn mit `-DTRUE` übersetzt wird (`p3.cc`)?

3.3.2 Fehler beim Binden

Beim **Bindevorgang** werden die angegebenen Objektdateien und Bibliotheken in ein ausführbares Programm zusammengefaßt. Wird das folgende Programm übersetzt,

```
extern int y;
int main () {
 int x=y+5;
}
```

dann werden keine Fehler berichtet (die Syntax ist korrekt), sondern der Binder `ld` beanstandet,

```
ld: Undefined symbol
 _y
Compilation failed
```

daß y nicht definiert ist. Durch die `extern` Anweisung wird explizit auf eine Definition in einer anderen Datei verwiesen. Ähnlich verhält es sich bei dem folgenden Programm.

```
#include <math.h>
int main() {
 double x=sqrt(1704.0);
}
```

Die Funktion `sqrt` ist nicht definiert, sondern nur in `math.h` deklariert. Entsprechend wird der folgende Fehler erzeugt:

```
ld: Undefined symbol
 _sqrt
```

Hier wurde vergessen, die mathematische Bibliothek durch den Schalter `-lm` dazuzubinden. Es stellt sich nun die Frage, wie es komfortabel möglich ist, einen Überblick über die Verwendung von symbolischen Namen zu erhalten. Hierbei nimmt einem das Programm `nm` einen Großteil der Arbeit ab. Der Aufruf `nm p5.o` erzeugt folgende Ausgabe

```
00000000 t ___gnu_compiled_cplusplus
         U ___main
00000000 T _main
         U _sqrt
00000000 t gcc2_compiled.
```

In der ersten Spalte stehen die Werte der symbolischen Namen. Die zweite Spalte gibt die Art des symbolischen Namens (3. Spalte) an. Die Buchstaben haben dabei folgende Bedeutung

```
t  definiertes lokales Text-Symbol
T  definiertes globales Text-Symbol
U  undefiniertes globales Symbol
```

Wird nm auf die mathematische Bibliothek angewendet, dann erscheint u.a. folgende Information

```
sqrt.o:
         U _SVID_libm_err
00000000 T _sqrt
```

Das Symbol sqrt wird in p5.o benutzt, dort aber nicht definiert (U in der entsprechenden Zeile). Hingegen wird in der Datei sqrt.o (befindet sich in der Bibliothek libm.a) das Symbol sqrt definiert. Der vollständige Aufruf zum Binden lautet daher g++ p5.o -lm bzw. CC p5.o -lm.

Die benutzerdefinierten Namen des C/C++ Programmes, also Variablen–, Funktions– und Typnamen, werden vom Übersetzer in symbolische Namen eines bestimmten Formats umgewandelt. So wird z.B. Variablennamen ein Unterstrich _ vorangestellt. Folgendes Beispiel verdeutlicht dieses Vorgehen.

```
class X {
 public :
  void do_something(int) { }
};

main() {
 X x;
}
```

Der Aufruf mit nm liefert

```
00000000 t ___gnu_compiled_cplusplus
         U ___main
00000000 t _do_something__1Xi
00000024 T _main
00000000 t gcc2_compiled.
```

Es wird ersichtlich, daß die Anzahl und Typ der Parameter der Methode X::do_something in dieses Format miteinfließt. Speziell für C++ –Programme steht das Programm nm++ zur Verfügung, welches teilweise dieses Format in die C++ –Bezeichner zurückinterpretiert.

```
00000000 t   ___gnu_compiled_cplusplus
         U   ___main
00000000 t   X::do_something
             [_do_something__1Xi]
00000024 T   _main
00000000 t   gcc2_compiled.
```

Das Programm nm von GNU kann das interne Format in den ursprünglichen benutzerdefinierten Namen zurückumsetzen. Mit dem Schalter -C werden C++ –konforme Namen generiert.

```
00000000 t __gnu_compiled_cplusplus
         U __main
00000000 t X::do_something(int)
00000024 T main
00000000 t gcc2_compiled.
```

Aufgabe 3.2

a Informieren Sie sich über die Möglichkeiten zur Ausgabe von symbolischen Namen mit den Programmen nm, nm++ und nm von GNU.

3.3.3 Bildschirmausgabe

Eine der einfachsten, aber oftmals wirkungsvollsten Methoden bildet die Ausgabe von Variableninhalten, Ablaufinformationen etc. zur Laufzeit auf den Bildschirm. Hier läßt sich meistens schnell überprüfen, ob das Programm das Gewünschte durchführt.

Werden viele Ausgaben zu schnell auf den Bildschirm ausgegeben, so daß diese nicht mehr zu lesen sind, liegt es nahe, diese mit dem Befehl > in eine Datei umzulenken (z.B. a.out > out).

Bei der Implementierung von Klassen in C++ bietet es sich an, eine Methode zu definieren, die den wesentlichen Teil (dieses wird in den meisten Fällen die Datenkomponente sein) auf den Bildschirm ausgibt.

```
class person {
  char* name;
  char* address;
  int   age;
```

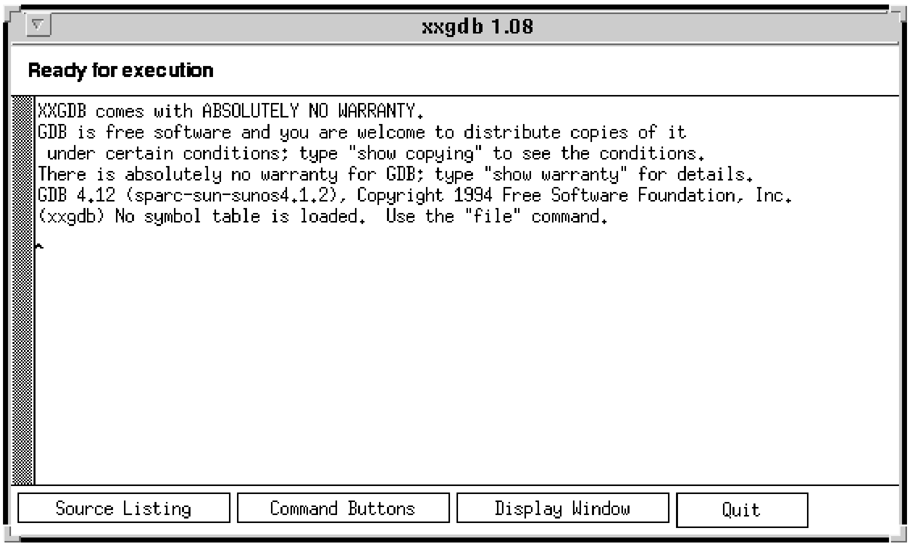


Abbildung 3.1: Der Debugger `xxgdb`

```
  public:
    void debug();
};

void person::debug() {
  cout << "name = " << name
       << "\naddress = "
       << address
       << "\nage = " << age
       << endl;
}
```

Durch den Aufruf der Methode `person::debug` wird somit die Datenkomponente auf den Bildschirm ausgegeben. Es läßt sich somit überprüfen, ob der Inhalt stimmt.

Soll die Ausgabe von bestimmten Bedingungen abhängig sein, dann ist es sinnvoll, die Testausgabe in `if` Anweisungen einzuschließen.

```
const int debug_me=1;

if (debug_me)
  cout << "Zeile 41 abgearbeitet."
       << endl;
```

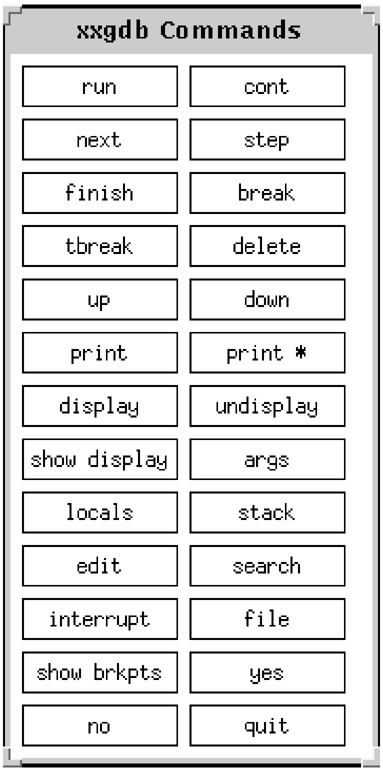
Wird die Ausgabe auf dem Bildschirm nicht mehr gewünscht, dann braucht die Variable `debug_me` nur auf `0` gesetzt werden.
Fernerhin ist es möglich, um die Ausführungszeiten nicht zu beeinflussen, keine `if` Anweisungen der C/C++ Sprachsyntax zu benutzen, sondern auf die entsprechenden Konstrukte des Präprozessors zurückzugreifen.

```
#ifdef DEBUG_ME
  cout << "Zeile 41 abgearbeitet."
       << endl;
#endif
```

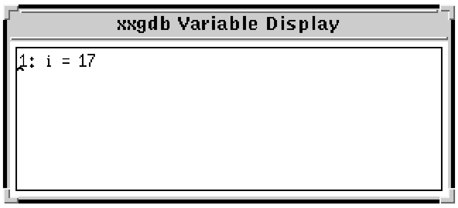
Ist die Ausgabe zu Testzwecken gewünscht, dann braucht das Programm nur mit der Schaltereinstellung `-DDEBUG_ME` übersetzt werden. Sonst wird diese Zeile nicht übersetzt.

3.3.4 xxgdb

Der Debugger `xxgdb` von GNU bietet die Möglichkeit, interaktiv zur Laufzeit in ein Programm einzugreifen. Um `xxgdb` in vollem Umfang benutzen zu können, müssen die einzelnen Quelldateien jedoch mit dem Schalter `-g` übersetzt



(a) Das `Command Buttons` Fenster



(b) Das Fenster zum Überwachen von Ausdrücken

Abbildung 3.2: Fenster des `xxgdb`–Debuggers

worden sein.

Weiterhin ist es ratsam, den Übersetzer `g++` von GNU dazu zu benutzen, da die Kompatibilität zwischen `CC` von SUN und dem GNU–Kompiler nicht hinreichend ist. Wird mit dem SUN–Compiler gearbeitet, dann sollte der Debugger `dbxtool`, der im nächsten Abschnitt beschrieben wird, benutzt werden.
Stürzt ein Programm ab, so wird in den meisten Fällen ein **Speicherabzug** in die Datei `core` (in der Regel mehrere MBytes lang) geschrieben. Diese Datei bietet nun die Möglichkeit, mit Hilfe des Debuggers die Stelle zu lokalisieren, an der das Programm abgestürzt ist.
Wird der Debugger gestartet, so kann durch die Angabe `core` der Speicherabzug eingelesen werden. Es läßt sich dann zum Beispiel feststellen, welcher Befehl als letztes ausgeführt wurde bzw. welchen Inhalt die Variablen zum Zeitpunkt der Programmterminierung hatten.
Es ist auch möglich, einen Speicherabzug zu erzwingen, in dem mit dem folgenden Befehl in einer Shell

`kill -QUIT` *ProzeßID*

das Programm terminiert wird.

Wird der Debugger gestartet, öffnet sich ein Fenster ähnlich wie in Abbildung 3.1. Hier können die verschiedenen Befehle per Tastatur eingegeben werden. Des weiteren besteht

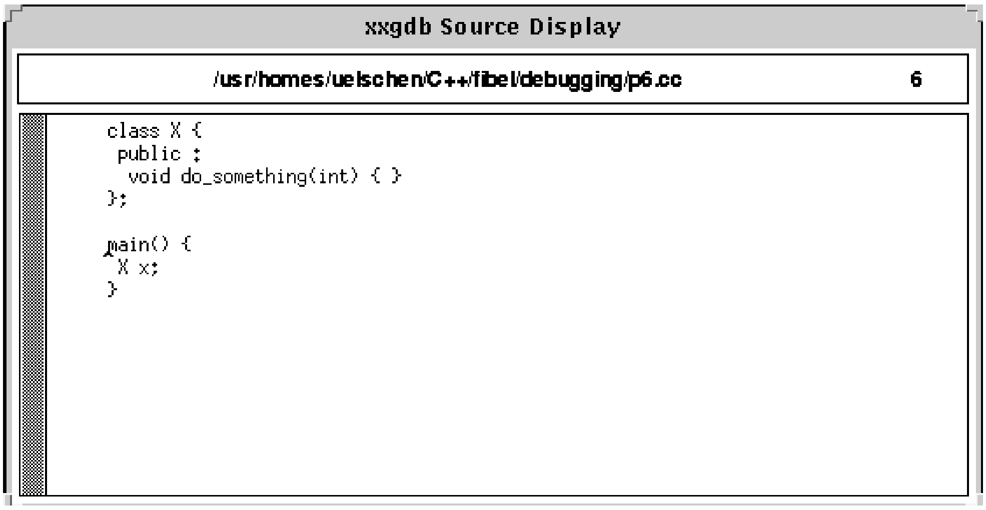


Abbildung 3.3: Das Quelldatei–Fenster

die Möglichkeit, eine Reihe von häufig benutzten Kommandos per Maus anzuwählen. Hierzu muß das `Command Buttons`–Fenster geöffnet werden (Abb. 3.2).

Nachfolgend werden die einzelnen Befehle kurz vorgestellt. Es handelt sich hierbei nur um eine geringe Auswahl der zur Verfügung stehenden Funktionen.

`run` startet das zu untersuchene Programm. Es ist möglich, dem Programm beim Start Argumente zu übergeben.

`cont` läßt ein unterbrochenes Programm mit der Ausführung weiter fortfahren. War die Ursache der Unterbrechung ein **Unterbrechungspunkt** (breakpoint), so kann als Parameter eine Zahl N angegeben werden, die bestimmt, nach wievielen Abläufen an dieser Stelle wieder unterbrochen werden soll.

`next` läßt das Programm bis zur nächsten Quellprogrammzeile weiterlaufen. Hierbei wird entgegen der `step`–Anweisung nicht in Prozeduren verzweigt. Durch Angabe einer Zahl N wird dieses dementsprechend häufig wiederholt.

`step` läßt das Programm bis zur nächsten Quellprogrammzeile weiterlaufen. Durch Angabe einer Zahl N wird dieses dementsprechend häufig wiederholt.

`finish` beendet die aktuelle Prozedur und gibt den Rückgabewert aus.

`break` setzt einen Unterbrechungspunkt in der angegebenen Zeile.

`tbreak` setzt einen Unterbrechungspunkt in der angegebenen Zeile. Dieser wird jedoch nach der ersten Unterbrechung automatisch wieder gelöscht (temporär).

`delete` löscht einen Unterbrechungspunkt.

`up` springt in der Aufrufhierachie der auf dem Stack abgelegten Prozeduren, abhängig von der aktuellen Position, eine Position höher.

`down` springt in der Aufrufhierachie, der auf dem Stack abgelegten Prozeduren, eine Position tiefer.

`print` gibt den angegebenen Ausdruck (ausgewertet) auf dem Bildschrim aus.

print* dereferenziert den angegebenen Ausdruck und gibt diesen anschließend auf dem Bildschirm aus.

display gibt den angegebenen Ausdruck im `Display Window` (Abb. 3.2) bei jeder Programmunterbrechnung aus.

undisplay löscht den angegeben Ausdruck aus der Liste der auszugebenen Ausdrücke.

show display entspricht `info display` und gibt die zu überwachenden Ausdrücke aus.

args entspricht `info args` und zeigt die Parameter beim Programmaufruf.

locals gibt die Variablen(–inhalte) der aktuellen Prozedur aus.

stack gibt den Stack der Prozeduraufrufe auf den Bildschirm aus.

edit (noch keine Funktion)

search sucht nach regulären Ausdrücken.

interrupt unterbricht das laufende Programm.

file lädt das angegebene Programm zum Debuggen.

show brkpts entspricht `info breakpoints` und gibt Informationen über die gesetzten Unterbrechungspunkte auf dem Bildschirm aus.

yes kann zum Beantworten von Entscheidungen seitens des Debuggers benutzt werden.

no ebenso.

quit beendet den Debugger.

Um den Quellcode anzeigen zu lassen, muß natürlich vorher das `Source Listing`–Fenster (Abb. 3.3) geöffnet worden sein. Ähnlich verhält es sich mit dem `Display Window` (Abb. 3.2). Werden zusätzliche Parameter verlangt, so müssen diese im Hauptfenster (Abb. 3.1) per Tastatur eingegben werden, z.B.

`run` *arg1 arg2*

Einige Befehle, z.B. `break`, `print` oder `display`, erlauben es, den Parameter im `Source Listing`–Fenster durch Anklicken zu spezifizieren.
Soll der Inhalt der Variablen `x` zum Beispiel ausgegeben werden, muß zuerst im Quelldatei–Fenster `x` mit Hilfe der Maus markiert und dann anschließend im `Command Button` Fenster `print` gedrückt werden.

Mit dem Befehl `help` ist es möglich, weitere Informationen zu den einzelnen Befehlen zu erhalten.
Des weiteren steht auch eine Kommandozeilenversion `gdb` für nicht–fensterorientierte Bedienoberflächen zur Verfügung.

Aufgabe 3.3
Informieren Sie sich über die grundlegenden Funktionen, die der GNU Debugger `xxgdb` bietet.

a Wie werden bedingte Unterbrechungspunkte definiert ?

b Wie lautet der Befehl zum Setzen von Variablen ?

c Wie lassen sich benutzereigene Befehle definieren ?

3.3.5 dbxtool

Der SUN–Debugger `dbxtool` bzw. die entsprechende Kommandozeilenversion `dbx` eignen sich ebenfalls zur interaktiven Untersuchung von Programmen zur Laufzeit.

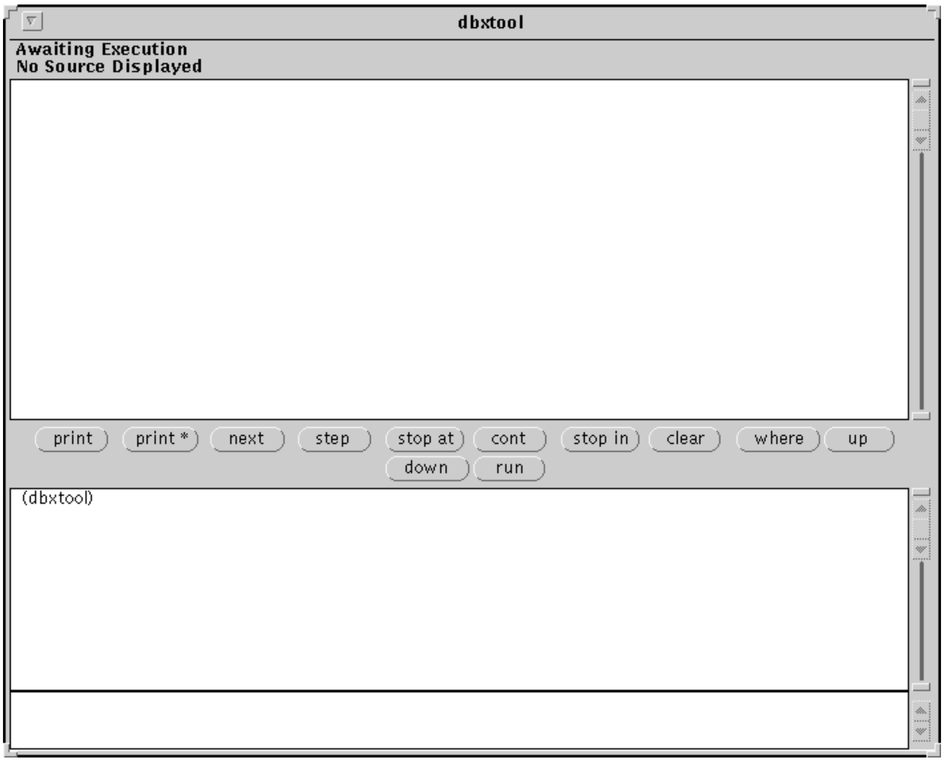


Abbildung 3.4: Der graphische Debugger `dbxtool` von SUN

Abbildung 3.4 zeigt das `dbxtool` Fenster. Das Fenster gliedert sich in vier Teile. Im oberen Teil wird der Quelltext angezeigt. In der Mitte befindet sich eine Knopfleiste mit einigen häufig zu gebrauchenen Befehlen. Im unteren Teil befindet sich, abgetrennt von der Überwachungsanzeige für Ausdrücke, das Eingabefenster, in dem die Befehle per Tastatur eingegeben werden können.
Nachfolgend werden einige Befehle kurz dargestellt.

`print` druckt den Wert von ein oder mehreren Ausdrücken auf dem Bildschirm aus.

`print*` dereferenziert bevor der Ausdruck ausgewertet und ausgegeben wird.

`next` läßt das Programm bis zur nächsten Zeile weiterlaufen, ohne in Unterprogramme zu verzweigen.

`step` läßt das Programm bis zur nächsten Zeile weiterlaufen. Es wird gegebenenfalls in Unterprogramme verzweigt.

`stop at` setzt einen Unterbrechungspunkt in der angegebenen (markierten) Zeile.

`cont` läßt das unterbrochene Programm weiterlaufen.

`stop in` setzt einen Unterbrechungspunkt in der angegeben Prozedur.

`clear` löscht den angegebenen Unterbrechungspunkt.

`where` gibt die aktuelle Position in der Aufrufhierarchie der Prozeduren auf dem Stack aus.

`up` springt in der Aufrufhierarchie eine Position nach oben.

`down` springt in der Aufrufhierarchie eine Position nach unten.

`run` startet das zu untersuchende Programm. Hierbei können diesem Argumente übergeben werden.

Die zur Verfügung stehende Funktionalität entspricht ungefähr der des oben erwähnten GNU –Debuggers. Auch `dbxtool` bietet eine Hilfefunktion `help` an, mit der weitere Informationen zu den einzelnen Funktionen erhältlich sind. Es soll nochmals darauf hingewiesen werden, daß die Quelldateien mit `CC -g` zu übersetzen sind, da sonst die benötigten Zusatzinformationen nicht zur Verfügung stehen.

Aufgabe 3.4
Informieren Sie sich über die Möglichkeiten, die der SUN Debugger `dbxtool` bietet.

a Wie lassen sich bedingte Unterbrechungspunkte realisieren ?

3.3.6 `trace`–Objekte

Sogenannte `trace`–Objekte erlauben es, das dynamische Verhalten von Programmen zu beobachten.
Wie bereits in den vorangegangenen Abschnitten beschrieben, existieren einige Hilfsmittel, wie z.B. der Debugger und die Bibliothek `dbmalloc`, die es ermöglichen, das Speicherverhalten zur Laufzeit zu beobachten.
Im folgenden wird gezeigt, wie durch die Benutzung von C++ –Konstrukten, dieses sind im wesentlichen Konstruktoren und Destruktoren (vgl. Lektion 6), das Laufzeitverhalten und das dynamische Verhalten von Objekten untersucht werden können.

In Abschnitt 3.3.3 wurde gezeigt, wie durch einfache Ausgaben auf dem Bildschirm der Programmablauf „visualisiert“ werden kann. Diese Möglichkeit soll im folgenden weiter vertieft werden. Insbesondere wird hierbei von Konstruktoren und Destruktoren Gebrauch gemacht.

```
class trace {
 public:
  trace() { cout << "Beginn."
                 << endl; }

 ~trace() { cout << "Ende."
                 << endl; }
};
```

Diese einfache Klasse `trace` besteht im wesentlichen nur aus einem Konstruktor und einem Destruktor.
Der Sinn solch einer Klasse wird erst ersichtlich, wenn man sich die Zeitpunkte, an denen Konstruktoren und Destruktoren aufgerufen werden, vor Augen hält. Der Konstruktor bzw. Destruktor einer Klasse wird bei der Definition bzw. der Zerstörung einer Instanz aufgerufen.

```
void kritisch() {
 trace xxx;

// tue etwas ...

}
```

Die einzige Aufgabe einer Instanz der Klasse `trace` ist es, seine Existenz anzuzeigen. Zu Beginn der Funktion `kritisch` wird eine Instanz `trace` erzeugt und, aufgrund des damit verbundenen (impliziten) Aufrufes des Konstruktors `Beginn.` auf dem Bildschirm ausgegeben.
Die Instanz `xxx` ist lokal in der Funktion `kritisch` definiert, was bedeutet, daß beim Rücksprung der Gültigkeitsbereich von `xxx` endet. Folglich wird der entsprechende Destruktor aufgerufen, der `Ende.` auf dem Bildschirm ausgibt.
Mit Hilfe dieser einfachen Methode ist es

möglich, den Programmablauf besser nachzuvollziehen.

Dieses Verfahren läßt sich auf die Überwachung des Speicherzugriffs (Allokation und Freigabe) übertragen und erweitern.
Für jedes Objekt, welches mit `new` erzeugt wurde, muß genau ein entsprechender `delete` Aufruf im Programm erfolgen. Ist der einem Objekt entsprechende Speicherbereich erst einmal freigegeben, so darf auf diesen auch nicht mehr zugegriffen werden, auch wenn sich der Speicherbereich seit der Freigabe nicht verändert hat. Fernerhin darf der Bereich des allokierten Speichers nur einmal freigegeben werden.
Wie läßt sich die korrekte Speicherfreigabe einfach überprüfen ?
Die grundlegende Idee ist, jeder Instanz eine eineindeutige „Seriennummer" zu geben. Hierdurch kann einfach überwacht werden, ob und wie häufig eine Instanz gelöscht wird.
Die Klasse `otrace` übernimmt diese Aufgabe.

```
class otrace {
  static int    count;
  int           serialnumber;
 public:
   otrace();
   otrace(const otrace&);
  ~otrace();

   otrace& operator=(const
                     otrace&);
};

int otrace::count=0;

inline otrace::otrace() {
 serialnumber=++count;
 cout << "Objekt "
      << serialnumber
      << " erzeugt." << endl;
}

inline otrace::~otrace() {
 cout << "Objekt "
      << serialnumber
      << " zerstoert." << endl;
}
```

```
inline otrace::otrace(const
 otrace&) {
  serialnumber=++count;
  cout << "Objekt "
       << serialnumber
       << " erzeugt." << endl;
}

inline otrace& operator=(const
 otrace&){
 return *this;
}
```

Als Beispiel sei fernerhin eine Klasse `string` gegeben, die überwacht werden soll. Diese wird um eine Datenkomponente `xxx` vom Typ `otrace` ergänzt.

```
class string {
// hier die Klassendefinition

// zusaetzlich im
// private Bereich
 otrace xxx;
};
```

Wird eine Zeichenkette instanziiert und damit der Konstruktor aufgerufen, werden auch die Konstruktoren der Datenelemente aufgerufen[1]. Der neuen Instanz wird eine Seriennummer zugewiesen, die beim Aufruf von Konstruktor und Destruktor auf dem Bildschirm mit einigen zusätzlichen Informationen ausgedruckt wird.
Die statische Deklaration von `otrace::count` sorgt für eine eineindeutige Zuweisung, d.h. die Instanzen vom Typ `string` werden fortlaufend durchnumeriert.
Ein Programmablauf könnte beispielsweise folgenden Ausdruck erzeugen:

```
Objekt 1 erzeugt.
Objekt 2 erzeugt.
Objekt 3 erzeugt.
Objekt 2 zerstoert.
Objekt 4 erzeugt.
Objekt 1 zerstoert.
```

[1] implizit, falls nicht explizit angegeben; vgl. Lektion 6

```
Objekt 3 zerstoert.
Objekt 4 zerstoert.
```

Eine fehlerhafte Speicherfreigabe ist nun anhand der erzeugten Ausgaben erkennbar.
Die Speicherfreigabe ist häufig bei sogenannten **Container–Klassen** ein Problem. So sei zum Beispiel eine Liste gegeben, in die Instanzen eines bestimmten Typs eingefügt werden können. Die Frage ist nun, wann die Objekte gelöscht werden sollen, d.h. wann der von ihnen allokierte Speicherbereich freigegeben werden soll. Eine Möglichkeit ist, beim Aufruf des Destruktors der Liste auch die Listenelemente zu löschen. Um das korrekte Verhalten nachzuvollziehen, eignet sich u.a. die oben vorgestellte `otrace` Klasse.

Aufgabe 3.5
Verdeutlichen Sie sich die Funktionsweise von Konstruktoren und Destruktoren (vgl. Lektion 6).

a Erweitern Sie die `trace` Klasse, so daß ein beliebiger (freiwählbarer) Text ausgegeben wird.

b Erweitern Sie die `trace` Klasse, so daß die Ausgabe auf einen beliebigen Ausgabestrom (z.B. in eine Datei) erfolgen kann.

c Welche Ausgabe wird durch folgende Anweisungen auf den Bildschirm gegeben ? Warum sollten keine dynamischen Instanzen der Klasse `trace` erzeugt werden ?

```
void kritisch() {
 trace* xxx=new trace();

 // tue etwas ...

}
```

3.4 Robuste Klassen

Während die in Abschnitt 3.3.6 beschriebenen Verfahren im wesentlichen den Programmverlauf anzeigen und der Programmierer diesen auf Fehler überprüfen muß, ist es natürlich auch möglich, die Verantwortung (teilweise) in die Klassen zu verlagern, d.h. die Instanzen überprüfen, ob eine Fehlersituation vorliegt.
Die hier beschriebenen Vorgehensweisen stammen aus [Dav94].

3.4.1 Das `asst`–Makro

Oftmals ist es gegeben, daß bestimmte Anweisungen nur dann ausgeführt werden dürfen, wenn vorher eine oder mehrere Bedingungen erfüllt sind. Bedingte Verzweigungen können in C/C++ durch `if then else` Anweisungen realisiert werden. Eine weitere Möglichkeit bietet das Präprozessor–Makro `assert` aus `assert.h`. Dieses definiert eine Funktion, die den als Argument übergebenen Ausdruck auswertet. Ist dieser null, wird eine Fehlermeldung mit Angabe der Datei und Zeilennummer, in der der Fehler aufgetreten ist, ausgegeben und mit `abort` das laufende Programm terminiert.
Beispiel:

```
int x=new int[9876543210];
assert(x);  /* Genug Speicher ? */
```

Solche Überprüfungen haben den Nachteil, daß sich der Ablauf des eigentlichen Programms verlangsamt. Ist ein Programm nach der Testphase für stabil befunden worden, ist es möglich, durch die Definition des Präprozessorsymbols `NDEBUG` vor der `#include <assert.h>` Anweisung die `assert`–Funktion in eine Nullanweisung umzuwandeln. Es reicht also, z.B. mit dem Schalter `-DNDEBUG` die betreffenden Dateien zu übersetzen.

Eine etwas komfortablere Möglichkeit bietet das folgende Makro:

```
#define REPORTLEVEL 0

#ifndef ASSERT
#define ASSERT(condition,severity) (\
```

```
 (condition) ? 1 :                  \
 (severity >= REPORTLEVEL) ?        \
 (cout << "assert failed : ("       \
                  #condition ") "\
       << __FILE__ << ":"           \
       << __LINE__ << endl), 0 : 0  \
)
#endif
```

Jeder Bedingung `condition` wird eine Gewichtigkeit `severity` zugeordnet. Fehler werden nur dann berichtet, wenn deren Gewicht größer oder gleich dem Schwellwert `REPORTLEVEL` ist.
Das folgende Beispiel zeigt eine mögliche Anwendung:

```
class student {
  long matnr;
 public:
  void set(long n);
};

void student::set(long n) {
 ASSERT(n>99999 && n<999999,0);
 matnr=n;
}
```

Aufgabe 3.6
a Erklären Sie die Syntax des `ASSERT`–Makros. Was bedeutet `#condition` ?

3.4.2 Objektzustände

Oftmals ist es möglich, daß die Adresse eines Objektes zwar gültig ist, der Zustand des Objektes aber ungültig.
Diese Situation tritt zum Beispiel dann auf, wenn im Konstruktor kein Speicher allokiert werden kann. Da auch kein Rückgabewert beim Konstruktoraufruf möglich ist, ist die Überprüfung einer erfolgreichen Instanziierung schwer möglich.

```
const int CHECK=42;


class anotherthing {
  char* mem;
  // dynamisch allociierter Speicher
  int signature;
 public :
  anotherthing();
 ~anotherthing();
  int checkSignature() const;
  char get(int) const;
};

anotherthing::anotherthing():
              signature(0),mem(0) {
 mem=new char[4321];
 if (mem==0)
  return;
 signature=CHECK;
}

anotherthing::~anotherthing() {
 delete mem;
 signature=0;
}

int anotherthing::checkSignature()
const {
 int ret=(signature==CHECK);
 if (ret==0)
  cout << "signature failed"
       << endl;
 return ret;
}

char anotherthing::get(int nr)
const {
 if (checkSignature())
  return mem[nr];
 else {
  cout << "memory access failed"
       << endl;
  exit(-1);
 }
}

int main() {
 anotherthing* y=new anotherthing;
 // tue etwas
 delete y;

 cout << y->get(77) << endl;
}
```

Ein häufiger Fehlerfall liegt vor, wenn auf Objekte zugegriffen wird, deren Speicherbereich bereits wieder freigegeben wurde. Der

Destruktor kann dazu benutzt werden, die Signatur auf einen vom Vergleichswert `CHECK` unterschiedlichen Wert zu setzen. Dadurch kann die beschriebene Fehlersituation aufgedeckt werden.

Nicht alle möglichen Fehlerquellen können durch die hier beschriebenen Konzepte aufgedeckt werden. Trotz alledem können robuste Klassen dazu dienen, frühzeitig Fehler im Entwicklungsprozeß zu entdecken.
Natürlich ist es möglich, diese Konzepte zu erweitern und miteinander zu kombinieren. Weitere Details sind dem Orginalartikel zu entnehmen.

3.5 Weitere Möglichkeiten

In [Koe94] werden von Andrew Koenig weitere Hinweise und Tips zum Debuggen gegeben. Nachfolgend werden diese kurz zusammengefaßt.

- Bevor der Programmtext verändert wird, sollte sichergestellt sein, ob überhaupt das Verhalten fehlerhaft ist. So sind z.B. Rundungsfehler bei Fließkommaoperationen nicht unüblich, da die interne Rechengenauigkeit begrenzt ist.

- Nachdem festgestellt wurde, daß das Verhalten fehlerhaft ist, sollte sichergestellt werden, daß im weiteren auch mit dem verbesserten Quelltext gearbeitet wird. Folgende Fehler sind häufig:

  – Das Programm wurde zwar verändert, allerdings ist vergessen worden, dieses neu zu übersetzen. Dieser Fall kann eintreten, wenn im Editor vergessen wurde, das Dokument (Quelltext) zu sichern.

  – Das Programm ist zwar neu übersetzt worden, allerdings wird eine alte (fehlerhafte) Version ausgeführt. Dieser Fall tritt dann ein, wenn verschiedene Pfade falsch gesetzt sind[2].

- Programme sollten in Komponenten (Modulen) entwickelt und getestet werden. So kann eine Verkettung mehrerer Fehler aus verschiedenen Modulen reduziert werden.

- Stehen **Klassenbibliotheken** zur Verfügung, sollten diese auch benutzt werden. Solche (kommerziellen) Bibliotheken sind in der Regel ausgereifter und weniger fehlerhaft. Des weiteren braucht das Rad nicht zum x-ten Mal neu erfunden zu werden.

- Klassen können Funktionen hinzugefügt werden, die überprüfen, ob Fehler vorliegen. Z.B. kann einem binären Baum Datentyp eine Methode `traverse` hinzugefügt werden, die den Baum durchläuft und überprüft, ob die Schlüssel der Elemente im linken Teilbaum kleiner der Schlüssel im rechten Teilbaum sind.

- Bei Klassen, die dynamisch Speicherplatz anfordern, sollte überprüft werden, ob die notwendigen **Konstruktoren** und **Destruktoren** definiert worden sind. Speicher der im Konstruktor allokiert wird, muß i.allg. spätestens im Destruktor wieder freigegben werden. Insbesondere ist zu überprüfen, ob der **Kopierkonstruktor** und der **Zuweisungsoperator** definiert wurden.

Aufgabe 3.7

a Betrachten Sie die folgende Definition des Zuweisungsoperators. Warum ist es wichtig, auf eine Selbstzuweisung zu überprüfen ?

[2] Ähnlich verhält es sich, wenn das eigene Programme `test` heisst. Unglücklicherweise existiert in UNIX™ Systemen bereits ein Programm mit dem Namen `test`, welches dann anstatt des eigenen aufgerufen wird. Also **nie** ein Programm `test` nennen!

```
something& something::operator=(
 const something& s) {
 if (this!=&s)
  // fahre mit den
  // Anweisungen fort ...

}
```

Bedenken Sie, daß `something` evtl. dynamisch Speicherbereich anfordern und freigeben muß. Wann hat dieses zu erfolgen ?

Lektion 4

Analyse und Design

In den nächsten Abschnitten werden einige Aspekte der objektorientierten **Analyse** (OOA) und des objektorientierten **Design**s (OOD) vorgestellt. Die weiteren Lektionen sind der objektorientierten **Programmierung** (OOP) gewidmet. Dieses beinhaltet die Umsetzung der in Analyse und Design gewonnenen Erkenntnisse. Die aufgeführten Konzepte und Ideen erheben keinen Anspruch auf Vollständigkeit. Prinzipiell sind verschiedene Ansätze zur Analyse und Design denkbar. Ziel soll es eher sein, einige grundlegende Konzepte darzustellen, um diese in den weiteren Lektionen anwenden zu können.

In der Analyse und dem Design bedient man sich oftmals einer graphischen Notation zur Beschreibung. Leider hat sich bisher keine einheitliche Bildsprache durchgesetzt. In der letzten Zeit haben sich im Prinzip drei (unterschiedliche) Methodiken und Notationen etabliert:

- der Ansatz von **Coad und Yourdon** (vgl. [CY91a] und [CY91b]),
- die Vorgehensweise von **Grady Booch** (vgl. [Boo91]) und
- die **Objektmodelltechnik** (OMT) von Rumbaugh (vgl. [RBP+93]).

In dieser Fibel wird die Notation von Rumbaugh benutzt.

Zur weiteren Vertiefung bezüglich objektorientiertem **Softwareengineering** sei auch auf das Buch [Mey90] von **Meyer** verwiesen.

4.1 Motivation

Wie bereits eingangs erwähnt, bildet die Softwareentwicklung einen Prozeß aus Analyse, Design und Implementierung, die aufeinander aufbauen.

Die wesentliche Schwierigkeit bei der Entwicklung größerer Programme liegt in der damit einhergehenden Komplexität. Obwohl die Gewichtung von Analyse und Design in komplexen Anwendungen deutlich stärker hervortritt, können die hier beschriebenen Ideen auch bei kleineren Projekten (z.B. Lektion 11) unterstützend und hilfreich sein.
Analyse und Design legen nicht nur die Spezifikation fest, sondern bilden die Grundlage für eine erfolgreiche Implementierung.

Die Analyse dient zur (statischen) Beschreibung des Problemfeldes unter Berücksichtigung objektorientierter Betrachtungsweisen. Das Design erweitert das in der Analyse gewonnene Schema um weitere (dynamische) Komponenten. Design ist in Hinblick auf eine spätere Implementierung ausgerichtet. Dieses betrifft insbesondere die Mensch–Maschine Schnittstelle. Oftmals bedient man sich in beiden Fällen graphischer Techniken.

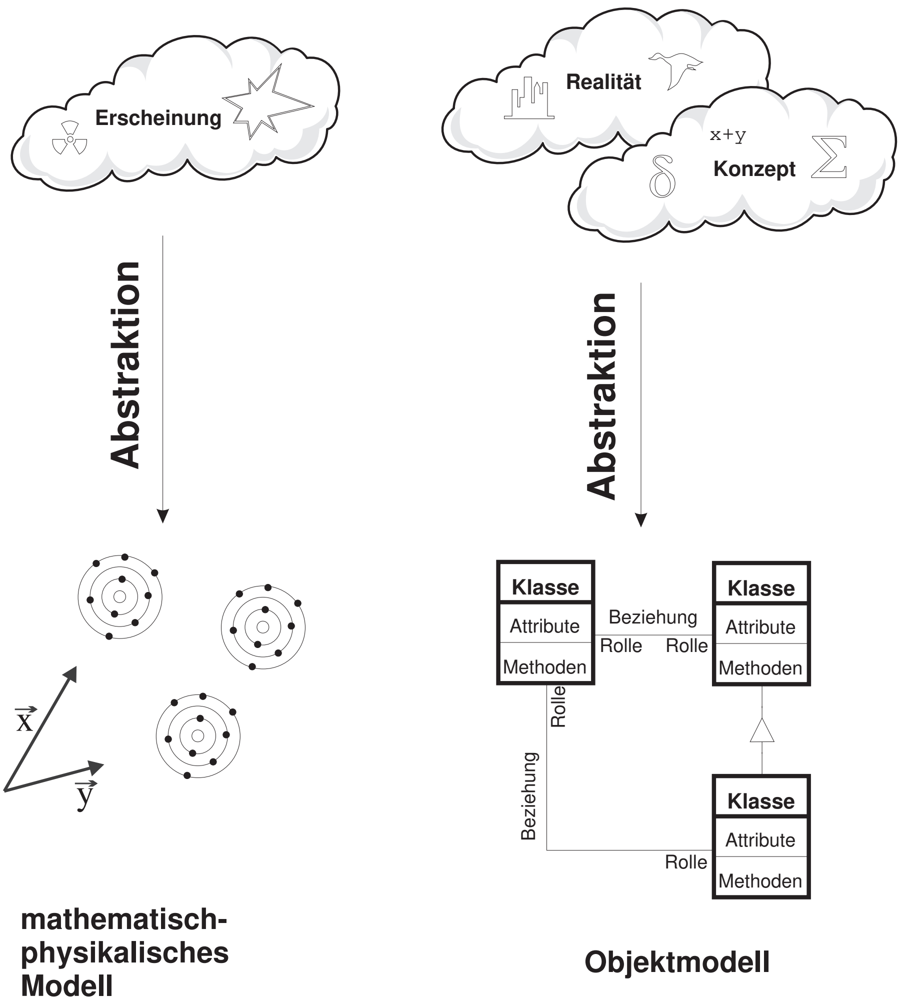


Abbildung 4.1: Modellierung

Die Übergänge zwischen Analyse und Design sind fließend. An dieser Stelle wird auf eine strikte Trennung verzichtet; es sei auf die genannte Literatur verwiesen. Die objektorientierte Programmierung befaßt sich mit der Umsetzung der in der Analyse und Design erworbenen Erkenntnisse in eine reale Programmiersprache (hier also C++).

Der objektorientierte Ansatz bietet eine Möglichkeit, die Komplexität bei der Softwareentwicklung handzuhaben. Diese geschieht u.a. durch folgende Prinzipien (vgl. [CY91a, Abschnitt 1.2]):

Abstraktion beschränkt die Modellbildung auf die relevanten (in Hinblick auf die Zielsetzung) Aspekte. **Prozedurale Abstraktion** beschreibt Prozeduren als Folge von Unterprozeduren (funktionale Dekomposition). **Datenabstraktion** definiert Typen in Form von zulässigen Operationen auf diesen.

Kapselung beschreibt die Trennung von Schnittstelle und realisierter Implementierung.

Vererbung beschreibt strukturelle Ähnlichkeiten von verschiedenen Klassen.

Meyer definiert in [Mey90, Seite 64] objektorientiertes Design so:

> Objektorientierter Entwurf ist die Entwicklung von Softwaresystemen als strukturierte Sammlungen von Implementierungen abstrakter Datentypen.

Objektorientierte Sprachen sind also weder etwas grundsätzlich neues noch die Lösung zu allen Problemen. Vielmehr handelt es sich um eine logische Fortführung von Konzepten und Ideen aus vorhandenen strukturierten Programmiersprachen.

Ziel objektorientierter Sprachen ist also ein einfacher und effektiver Weg vom Problem zur Lösung; das ist i.allg. ein Programm, welches die gestellte Aufgabe löst.

4.2 Modellbildung

Das fundamentale Prinzip im objektorientierten Softwareentwurf ist die **Modellbildung**. Ein Modell ist eine **Abstraktion** von der (realen) Welt. Es verzichtet auf für die Fragestellung redundanten Details und bildet nur die wesentlichen Aspekte nach.
Abbildung 4.1 zeigt die Vorgehensweise in der Physik und im objektorientierten Softwareengineering in der Informatik. Die Physik versteht sich als Wissenschaft zur Erklärung der *Erscheinungswelt* oder meßbaren Welt. Um eine physikalische Erscheinung, z.B. den radioaktiven Zerfall in der Natur oder den fallenden Apfel vom Baum, zu erklären und zu verstehen, bedarf es der Reduzierung der Komplexität der Realität durch ein Modell. Oftmals wird sich hierzu der Mathematik bedient, die die formalen Werkzeuge bereithält. Ein zentrales Modell in der Physik stellt das *Bohrsche Atommodell* dar. Kräfte auf Körper können durch ein Konzept aus der Mathematik, nämlich durch die Einführung sogenannter *Vektoren*, nachgebildet werden.
Wichtig ist es, bei der Modellierung zu überprüfen, ob Erkenntnisse und Ergebnisse aus dem Modell auf die Realität noch übertragbar sind. Bei der Modellbildung muß die Übertragbarkeit also immer im Vordergrund stehen; ohne sie wird das Modell wertlos.
Im objektorientierten Softwareengineering in der Informatik wird, ähnlich wie in der Physik, als zugrundeliegendes Prinzip die Modellbildung als Entwurfstechnik für Software verwendet. Das rechte Diagramm in Abbildung 4.1 zeigt diesen Zusammmenhang. Ausgehend von der realen bzw. konzeptionellen Welt wird

der wesentliche Ausschnitt abstrahiert und in ein **Objektmodell** gefaßt. Um das Modell zu beschreiben, bedient man sich oftmals einer graphischen Notation.
Als Beispiel sei ein Arztabrechnungssystem gegeben. Um ein Programm zu entwickeln, welches die Anforderungen an solch ein System erfüllt, ist bei einem objektorientierten Ansatz der Ausschnitt aus der Realität (Patienten, Arneimittel, Krankheiten, Verwaltungsvorgänge, ...) in ein Modell zu fassen.
Ein weiteres und offensichtliches Betätigungsfeld objektorientierter Ansätze liegt im Bereich der computergestützten Simulation vor, da hier eine abstrahierte Abbildung der Realität im Computer untersucht, d.h. simuliert, wird. Die Übernahme von wichtigen Konstrukten aus Simula in C++ (vgl. Abschnitt 1.1) scheint also nur logisch.
Ein Unterschied bezüglich der Zielsetzung der Modellierung in der Physik und der Informatik sei noch kurz erwähnt. In der Physik wird versucht, die (reale) Welt zu erklären. Hingegen steht in der Informatik, wenn man von der Simulation absieht, die Lösung eines konkreten Problems im Vordergrund (ingenieurmäßiger Gesichtspunkt).

In der *Objektmodelltechnik* von Rumbaugh wird das Systemmodell in drei Teilmodelle aufgegliedert:

Das **Objektmodell** beschreibt statische, strukturelle und datenbezogene Aspekte,

das **dynamische Modell** beschreibt zeitliche, verhaltensmäßige und steuerungsbezogene Aspekte,

das **funktionale Modell** beschreibt Übergangs–, Funktions– und Transformationsaspekte.

In dieser Fibel wird nur auf das **Objektmodell** eingegangen. Das dynamische Modell beschreibt zeitliche Zustandsabfolgen. Dieses kann anhand von Zustandsdiagrammen verdeutlicht werden. Das funktionale Modell hingegen drückt aus, wie einzelne Zustände erreicht werden.

4.3 Objektmodell

Wie bereits eingangs geschildert, werden in der OOA das Problemfeld untersucht und die relevanten Aspekte modelliert. Hierbei wird sich oftmals einer graphischen Notation bedient.
Die folgenden Abschnitte geben eine kurze Einführung in die objektorientierte Gedankenwelt. Die verschiedenen Konzepte werden anhand der OMT von Rumbaugh dargestellt.

4.3.1 Objekt und Klasse

Die wesentlichen Gegenstände in der objektorientierten Gedankenwelt bilden **Objekte** und **Klassen**. Objekte stellen Abstraktionen von realen oder konzeptionellen Gegenständen im betrachteten Problemkreis dar. **Klassen** fassen Objekte gleicher Art zusammen.
Bsp.: das Buch mit dem Titel *The C++ Programming Language*, *Bedienungshandbuch zum Drucker MPS 1000* oder *Der Westen der USA* ist ein Objekt aus der Realität. Diese drei und weitere Objekte lassen sich in der Menge, also der **Klasse**, der Bücher zusammenfassen.
Objekte aus der konzeptionellen Welt könnten zum Beispiel die Funktionen $f(x) = x^3 + 7x^2$ und $f(x) = 3x^2 - 45x$ sein. Diese ließen sich in der Klasse der Polynome gruppieren.
Abbildung 4.2 (links oben) zeigt die Graphik für eine Klasse. Klassen werden durch Rechtecke dargestellt. Im Inneren ist der Klassenname zu finden. Einzelne Objekte werden durch ein Rechteck mit abgerundeten Ecken notiert. Der Klassenname wird dabei in Klammern gesetzt. Da jedoch i.allg. Klassen modelliert werden, ist der Gebrauch von Objektgraphiken selten. Sie

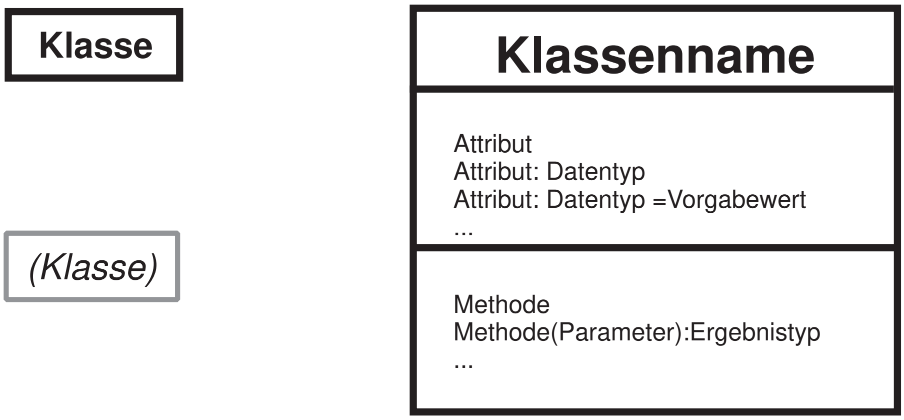


Abbildung 4.2: Klasse und Objekt

eignen sich aber, um sogenannte **Szenarien**, also Beispiele, zu veranschaulichen. Abbildung 4.2 (links unten) zeigt die Notation für ein Objekt.

Leider wird der Begriff *Objekt* und *Klasse* oftmals in der Literatur ein wenig inkonsistent verwendet. Viele Autoren bezeichnen mit *Objekt* auch die zugehörige *Klasse*. Dieser Umstand wird noch dadurch unnötig kompliziert, daß auch von *Objektinstanzen* und *Objektklassen* gesprochen wird (vgl. Rumbaugh [RBP+93]). In dieser Fibel werden die Begriffe wie folgt verwendet:

Objekte sind Modelle einzelner Gegenstände. Bezüglich objektorientierter Programmiersprachen wird auch oftmals von **Instanzen** gesprochen. Einer Instanz ist ein genau definierter Abschnitt im Speicher zugeordnet.

Klassen fassen im üblichen mathematischen Sinn Objekte zu Mengen zusammen, die (strukturelle) Gemeinsamkeiten aufweisen. In (objektorientierten) Programmiersprachen stellen Klassen **Typen** dar.

Konsequenterweise müsste es daher eigentlich *klassenorientierte* Analyse und *klassenorientiertes* Design heißen, da im allgemeinen Klassen und keine einzelnen Objekte modelliert werden. Diese Begriffe sind aber unüblich und werden nachfolgend auch nicht weiter verwendet.
Die Programmiersprache C++ stellt zur Definition von Klassen die Schlüsselwörter `struct` und `class` zur Verfügung. Fernerhin existiert eine Reihe vordefinierter atomarer Typen, so z.B. der Typ `int`, welcher die Klasse, d.h. die Menge, der ganzen Zahlen beschreibt.

4.3.1.1 Attribute

Ein **Attribut** ist eine einzelne Eigenschaft einer Klasse. Attribute werden im Objekt mit Werten belegt. Auf das obige Beispiel bezogen, könnte die Klasse *Buch* durch die Eigenschaften *Titel* und *Autor* charakterisiert werden. Ein Objekt hätte also den *Titel* mit dem Wert *The C++ Programming Language* und die Eigenschaft *Autor* mit dem Wert *Bjarne Stroustrup*.
Im Sprachgebrauch der (objektorientierten) Programmiersprachen sind Attribute klassenspezifische Variablen. Diese werden in C++ direkt unterstützt.

Im Objektdiagramm werden die Attribute einer Klasse im mittleren Feld aufgeführt (vgl. Abbildung 4.2 rechts). Optional kann ein Typ und ein Vorgabewert spezifiziert werden. Oftmals wird es aber sein, daß auf die einzelnen Attribute und Methoden (siehe nächster Abschnitt) in der Darstellung aus Übersichtsgründen verzichtet wird. Statt dessen wird das einfachere Diagramm (vgl. Abbildung 4.2 links) verwendet.
Objekte bzw. Klassen werden jedoch nicht nur durch die (statischen) Eigenschaften und Beziehungen beschrieben, sondern insbesondere durch ihre Funktionalität.

4.3.1.2 Methoden

Die **Methoden** oder auch **Dienste** eines Objektes verändern deren Attribute oder Beziehungen (siehe nächster Abschnitt). So könnte z.B. der Dienst *Andere Straße* das Attribut *Straße* eines Objektes der Klasse *Person* auf einen neuen Wert setzen.

Stroustrup [Str91, Seite 375] regt u.a. folgende Gedanken zur Bestimmung der Methoden (oder Operationen) an:

- Wie wird ein Objekt der betrachteten Klasse konstruiert ?
- Welches ist die minimale Menge von Operationen, die benötigt werden ?
- Welches sind die wichtigsten Operationen, die zusätzlich benötigt werden (z.B. zum Testen oder zur Ausgabe) ?

Methoden lassen sich in folgende Kategorien fassen:

Erzeugung/Zerstörung Methoden, die zur **Instanziierung** (Konstruktion) und zur Destruktion dienen.

Auswahl Methoden, die Attribute oder Beziehungen unverändert lassen (Lesezugriff).

Modifikatoren sind Methoden, die (einzelne) Attribute oder Beziehungen verändern (Schreibzugriff).

Umwandlungsoperationen transformieren ein Objekt in ein Objekt einer anderen Klasse.

Iteratoren sind Methoden, die hauptsächlich bei Sammlungen anzufinden sind und den Zugriff auf einzelne Objekte erlauben.

In C++ werden Methoden zur Konstruktion und zur Destruktion besonders behandelt.
Die Methoden definieren die Schnittstelle einer Klasse. In C++ werden diese durch klassenspezifische Funktionen bzw. Prozeduren bereitgestellt. Der Zugriff auf einzelne Attribute wird (sollte) durch geeignete Methoden erfolgen. Dieser Mechanismus wird auch oftmals als **Kapselung** bezeichnet und wird in C++ durch die Schlüsselwörter `public`, `protected` und `private` unterstützt.
Im Objektmodell (vgl. Abbildung 4.2) werden die Methoden im unteren Feld aufgeführt. Optional können Parametertypen und Ergebnistyp angegeben werden.

Methoden und Zugriffskontrolle sind fundamentaler Unterschied zu herkömmlichen prozeduralen Sprachen wie z.B. Modula–2 oder C.

4.3.2 Beziehungen

Beziehungen verbinden zwei oder mehrere Klassen miteinander. Beziehungen werden oftmals auch als **Assoziationen**, **Verknüpfungen** oder **Relationen** bezeichnet. In dieser Fibel wird der erste Begriff bevorzugt. Abbildung 4.3 zeigt eine Beziehung zwischen zwei Klassen. Beziehungen sind i.allg. binär, in einigen wenigen Fällen auch ternär [1]. Eine Beziehung wird durch eine Linie, versehen mit einem Namen, im Klassendiagramm dargestellt. Eine

[1] Höherwertige Beziehungen sind schwer handhabbar.

Abbildung 4.3: Beziehung zwischen zwei Klassen

Beziehung läßt sich in eine Hin– und in eine Rückbeziehung aufteilen. Dieses ist insbeondere für die Kardinalität, die im folgenden Abschnitt besprochen wird, wichtig.
Ein weiter unten aufgeführten Beispiel zeigt, daß auch eine Klasse mit sich selber in Beziehung stehen kann.

Beziehungen werden in C++ nicht direkt, d.h. durch spezielle Konstrukte oder Schlüsselwörter, unterstützt. Statt dessen können sogenannte Zeiger und Referenzen benutzt werden.

4.3.2.1 Kardinalität

Die **Kardinalität** oder auch **Multiplizität** gibt an, wieviele Objekte eine Klasse miteinander verknüpft sind. Im Prinzip sind die folgenden Fälle zu unterscheiden:

1:1 bedeutet, daß ein Objekt der Klasse 1 mit einem Objekt der Klasse 2 in Beziehung steht.

1:n bedeutet, daß ein Objekt der Klasse 1 mit einer beliebigen Anzahl von Objekten der Klasse 2 in Beziehung steht.

n:n bedeutet, daß beliebig viele Objekte der Klasse 1 mit beliebig vielen Objekten der Klasse 2 in Beziehung steht.

Abbildung 4.4 zeigt die Kardinalität im Klassendiagramm anhand eines Beispiels. Hier zeigt sich auch, daß eine Beziehung automatisch immer in zwei Richtungen existiert: ist ein Objekt x mit einem Objekt y verknüpft, so folgt daraus, daß auch das Objekt y mit dem Objekt x verknüpft ist.
Das folgende Beispiel in Abbildung 4.4 zeigt diesen Zusammenhang.

> Ein PKW gehört genau einer Person. Hierbei handelt es sich also um eine 1:1 Beziehung. Umgekehrt kann eine Person der Besitzer einer beliebigen Anzahl von PKW sein. Dieses wird durch eine 1:n Beziehung ausgedrückt.

Um die Kardinalität einer Beziehung zu bestimmen, ist also immer in Hin– und Rückbeziehung zu unterscheiden.
Um das Objektmodell visuell nicht unnötig zu komplizieren, sind einige Vereinfachungen möglich. Endet eine Beziehung mit einem schwarzen Punkt (z.B. wie in Abbildung 4.5), so deutet dieses auf eine 1:n Beziehung hin. Fehlt dieser Punkt, so handelt es sich um eine 1:1 Beziehung.

4.3.2.2 Rolle

Neben der Bezeichnung der gesamten Beziehung zwischen (zwei) Klassen ist es üblich, zusätzlich oder anstatt des Beziehungsnamen die **Rolle** der Klasse anzugeben. Die Rolle einer Klasse gibt an, welche Funktion diese bezüglich der anderen Klasse in der Beziehung ausführt. Die Rolle jeder Klasse in einer Beziehung ist an den Enden (vgl. Abbildung 4.5) zu finden. Eine Klasse hat i.allg. verschiedene Rollen, nämlich in jeder Beziehung eine andere. In dem auf-

Abbildung 4.4: Kardinalität einer Beziehung

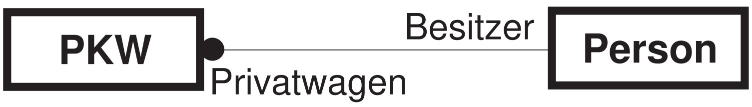


Abbildung 4.5: Rolle von Klassen in einer Beziehung

geführten PKW–Beispiel lassen sich die Rollen, wie in Abbildung 4.5 gezeigt, definieren.

> Ein PKW ist ein Privatwagen (Rolle) genau einer Person. Eine Person ist der Besitzer (Rolle) einer beliebigen Anzahl von PKW.

Das folgende Beispiel zeigt, wie eine Klasse mit sich selber in Beziehung stehen kann (vgl. Abbildung 4.6).

> Eine Person (Vater–Rolle) steht mit eine beliebigen Anzahl von (anderen) Personen in Beziehung. Eine Person (Sohn–Rolle) ist mit genau einer (anderen) Person verknüpft.

Das Rollenkonzept ist ein Werkzeug, um das Objektmodell aussagekräftiger zu gestalten. Einen direkten Niederschlag in der Programmiersprache C++ hat es nicht gefunden. Es ist jedoch möglich, den Rollennamen als Bezeichner für die Beziehung, dieses wird i.allg. ein Zeiger in C++ sein, zu wählen.

4.3.2.3 Attributierte Beziehungen

Es ist oftmals nötig, an eine Beziehung ein oder mehrere Attribute zu binden. In solch einem Fall wird von einer **attributierten Beziehung** gesprochen. Abbildung 4.7 zeigt dieses an einem einfachen Beispiel:

> Ein Benutzer hat Zugriff auf eine bestimmte Menge von Rechnern. Diese Verknüpfung hat das Attribut *Kennwort*, da es in UNIX™ Systemen üblich ist, Zugriffsberechtigungen durch Kennwörter zu vergeben.

Es zeigt sich, daß es nicht möglich ist, das Atrribut der Beziehung in eines der beiden Klassen zu verlagern. Im ersten Fall (siehe Abbildung 4.8) hätte der Benutzer auf jedem Rechner das gleiche Kennwort, was zwar erlaubt ist, aber eine unnötige Einschränkung bedeutet. Im anderen Fall gäbe es auf jedem Rechner nur ein einziges Kennwort für alle Benutzer, was dem eigenlichen Sinn der Zugriffskontrolle zuwiderläuft. Im Beziehungsdiagramm wird die attributierte Beziehung durch ein Rechteck mit der Auflistung der Attributen ergänzt.

4.3.2.4 Ordnung

Soll qualifiziert werden, daß die Elemente in einer Beziehung in der Reihenfolge bezüglich einer Ordung unterscheidbar sind, so kann die Be-

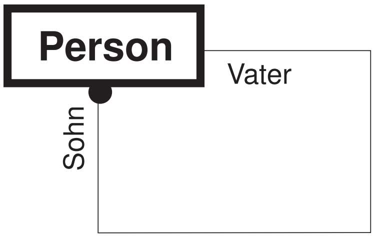


Abbildung 4.6: Beziehung einer Klasse zu sich selber

Abbildung 4.7: Attributierte Beziehung

ziehung durch einen Zusatz (siehe Abbildung 4.9) ausgezeichnet werden.

4.3.2.5 Qualifikation

Beziehungen können qualifiziert werden. Hierdurch ist es möglich, weitere Semantik in das Objektmodell zu integrieren. Die Qualifikation in Zusammenhang mit einem Objekt der Klasse 1 bestimmt das Objekt der Klasse 2 (Abbildung 4.10).

4.3.2.6 Aggregationen

Eine Sonderform der Beziehungen ist die Aggregation oder auch **ist–Teil–von** Beziehung. Dieser **Enthaltensein** Zusammenhang wird auch als **Sammlung** bezeichnet. Im Objektdiagramm wird diese spezielle Beziehung durch eine Raute (vgl. Abbildung 4.11) dargestellt.
In objektorientierten Programmiersprachen werden oftmals **Container** Klassenbibliotheken angeboten, die häufig benötigte Aggregationen definieren. Hierzu gehören z.B. Mengen, Listen, Bäume und Tabellen.

4.3.3 Vererbung

Eine besondere Beziehung ist die **ist–ein** Beziehung. Ein Objekt der Klasse 1 **ist-ein** Objekt der Klasse 2 (vgl. Abbildung 4.12). Es soll damit zum Ausdruck gebracht werden, daß eine Klasse Eigenschaften und Funktionalitäten einer anderen Klasse übernimmt. Dieses wird auch oftmals als **Generalisierung–Spezialisierung** Kontext bezeichnet. Eine Klasse kann eine spezialisierte Form einer allgemeineren (generalisierten) Klasse sein. So ist z.B. die Klasse aller Automobile die generalisierte Form der spezialisierten Klassen PKW und LKW.
Dieses wird gemeinhin auch als **Vererbungsprinzip** bezeichnet.
Die Vererbung wird in C++ (aber nicht in C) direkt unterstützt. Sie wird bei der Deklaration einer Klasse explizit angegeben.
Vorteil der Vererbung ist die Möglichkeit, exi-

Abbildung 4.8: Verlagerung des Attributes einer Beziehung

Abbildung 4.9: Beziehung mit Ordnung

stierende Software weiter– und wiederzuverwenden, indem vorhandene Klassen erweitert und/oder modifiziert werden können.
Des weiteren sei angemerkt, daß Vererbung die Übernahme von strukturellen Eigenschaften meint und nicht Belegung von Attributen mit geerbten Datenwerten eine generalisierten Klasse (es werden Klasseneigenschaften vererbt !).
Aus dem Bereich der Betriebssysteme ergibt sich folgendes interessantes Beispiel: Dateien lassen sich als verallgemeinerte Klasse der spezialisierten Klassen Datendatei, Programmdatei und Verzeichnis auffassen. Objekte der Klasse Verzeichnis sind Sammlungen von Dateien.
Ein anderes Beispiel ist in Abbildung 4.13 dargestellt. Im Objektdiagramm wird die Vererbung mit Hilfe eines Dreiecks dargestellt (vgl. Abbildung 4.12). Die in den letzten Abschnitten beschriebenen Ideen sollen anhand eines etwas größeren Beispiels noch einmal vorgeführt werden. Abbildung 4.14 zeigt (vereinfacht) den Bereich der Benutzerverwaltung in einem UNIX™ Rechennetz[2]. Aus Übersichtsgründen wurde auf eine vollständige Auflistung der Attribute und Methoden verzichtet.

> Ein Benutzer hat auf einer Menge von Rechnern eine Zugriffsberechtigung. Auf jedem Rechner ist ein Kennwort frei wählbar. Eine Gruppe faßt eine Menge von Benutzern auf einem Rechner zusammen. Neben Benutzern und Gruppen sind Dateien auf einem Rechner lokalisiert. Eine Datei ist die generalisierte Klasse der beiden Klassen Programmdatei und Verzeichnis. Jede Datei gehört genau einem Benutzer. Ein Verzeichnis ist eine Aggregation von Dateien. Jeder Benutzer hat ein spezielles Heimverzeichnis.

4.4 Vorgehensweise

Bei der objektorientierten Analyse und Design sind also i.allg. die folgenden Aufgaben durchzuführen:

- **Auffinden der Klassen und Objekte**
- **Definieren der Attribute**

[2] In dem Modell ist die Beziehung zwischen Datei und Gruppe, die die Zugriffsrechte definiert, nicht aufgeführt.

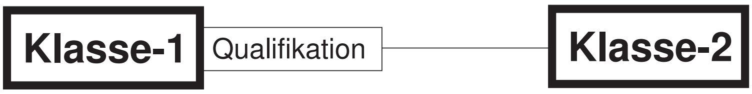


Abbildung 4.10: Beziehung mit Qualifikation

Abbildung 4.11: Beziehung als Aggregation von Objekten

- **Definieren der Methoden**
- **Identifizieren der Beziehungen**

In den nächsten Lektionen werden die einzelnen Aktivititäten konkretisiert. Es zeigt sich, daß viele Konzepte in C++ (fast) direkt übernommen werden können. Ein gutes und durchdachtes Objektmodell verkürzt also die Implementierung und hilft konzeptionelle Fehler bei der Softwareentwicklung auszuschliessen.

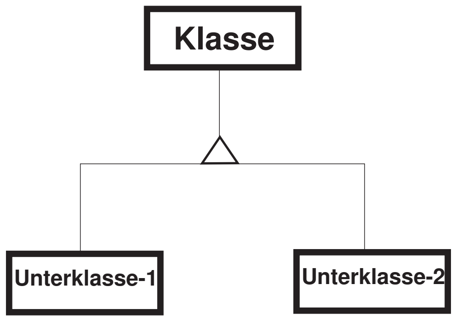


Abbildung 4.12: Vererbung

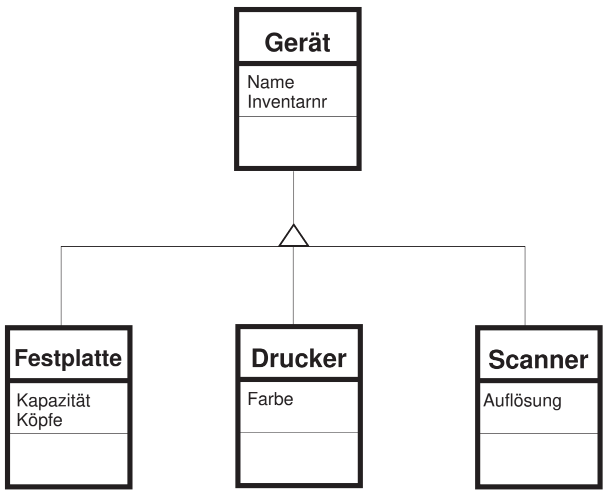


Abbildung 4.13: Beispiel zur Vererbung

Bjarne Stroustrup bemerkt in [Str91, Seite 374] folgende interessante Parallele zur natürlichen Sprache:

> *Substantive* in einem Satz lassen sich als Objekte bzw. als Klasse auffassen. Die im Satz vorkommenden *Verben* sind Methoden auf diesen.

Dieses kann jedoch nur als grobe Richtlinie gelten; Ausnahmen sind offensichtlich.

Aufgabe 4.1
Versuchen Sie Probleme unter objektorientierten Gesichtspunkten zu begreifen und zu formulieren. **Programme sind Modelle der realen Welt.**

a Modellieren, d.h. Analysieren und Entwerfen Sie das Verhalten von Autos vor einer Ampel mit bestimmten Rot–Grün–Intervallen.

b Modellieren Sie ein UNIX™ Rechennetz. Bestimmen Sie die verschiedenen Objekte (Brücke, Drucker, . . .) und ihre Beziehungen untereinander.

c Entwerfen Sie ein Modell zur Simulation von Warteschlangen in einer Bank. In der Bank befinde sich ein Schalter, an den Kunden herantreten und ihre Bankangelegenheiten erledigen können. Ist der Schalter gerade besetzt, so haben die Kunden in der Reihenfolge ihrer Ankunft zu warten. Wird der Schalter frei, so tritt der am längsten Wartende an den Bankbediensteten heran.

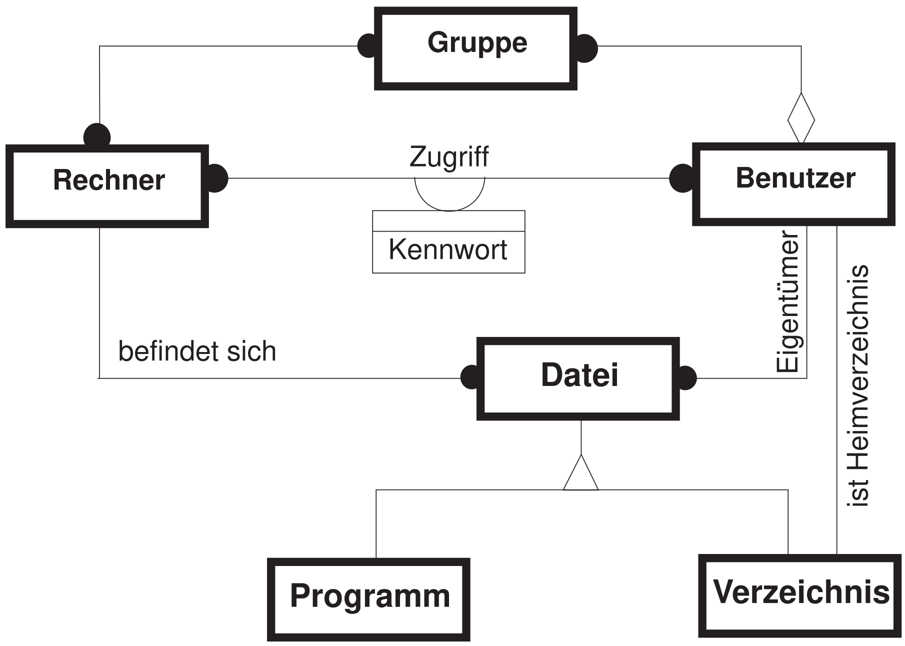


Abbildung 4.14: Benutzerverwaltung in UNIX™ Systemen

Lektion 5

Einführung

In den nächsten Abschnitten wird eine kurze Einführung in die Programmierung mit C++ gegeben.

5.1 Deklaration und Definition

ISO 3.1 Bevor ein **Name** (oder auch **Identifizierer** oder **Bezeichner**) für ein Objekt benutzt werden kann, muß dieser dem C++ –Übersetzer bekannt gegeben, d.h. **deklariert**, werden.
Hierzu wird dem Namen ein **Typ** zugewiesen, der dann die möglichen **Operationen**, die auf das zum Namen gehörende Objekt ausgeführt werden können, spezifiziert.
Wird das Objekt zum Zeitpunkt der Deklaration erzeugt, d.h., wird Speicherbereich bereitgestellt, spricht man von einer **Definition**.

```
int c=2;
char *a="C++ macht Spass";
unsigned long x;
extern int r;
```

Nur bei der letzten Deklaration handelt es sich um keine Definition, da durch das Präfix `extern` ausdrücklich kein Speicherbereich für das Objekt `r` zur Verfügung gestellt wird.
Wird dem Objekt bei der Definition ein eindeutig bestimmter Wert zugewiesen, handelt es sich hierbei um **initialisierte** Objekte. Nicht initialisierte Objekte (im Beispiel `x`) besitzen einen nicht definierten Wert. .

Eine wichtige und häufig benutzte Typklasse bilden die sogenannten **Zeigertypen**. Ein Zeiger enthält die Adresse eines Objektes im Speicher. Zeiger werden mit Hilfe des Dereferenzierungsoperators `*` definiert, z.B. ISO 8.3.1

```
int* ptr;
  /* Zeiger auf Ganzzahl */
```

Mit Hilfe des Adreßoperators `&` ist es möglich, Zeigern bestimmte Objekte zuzuweisen.

```
int i=1704;
*ptr=&i;
  /* ptr zeigt auf i */
```

Eingeschränkt sind auf Zeigern auch Operationen wie z.B. Erhöhen und Erniedrigen möglich.

```
char *b=a+1;
  /* b zeigt auf '+' */
```

In den oben genannten Beispielen wird ein Zeiger `a` auf eine Zeichenkette "`C++ macht Spass`" definiert und initialisiert. Genauer zeigt `a` auf das erste Zeichen `'C'`. Die Zeichenkette wird mit einem `'\0'`–Zeichen abgeschlossen, welches automatisch vom Übersetzer bei mit " begrenzten Zeichenketten angefügt wird. Das Ende von Zeichenketten läßt sich also einfach feststellen. Es sei darauf hingewiesen, daß Zeichenketten keinen eigenen Typ bilden, sondern, wie beschrieben, durch Zeiger auf das erste Element vom Typ `char` ausgedrückt werden. Folglich sind standardmäßig auch keine anderen Operationen als jene auf Zeigern zugelassen.

Eine weitere wichtige Typklasse bilden die sogenannten **Referenzen**. Im Prinzip handelt es sich hierbei um alternative Namen (Alias) für Objekte.

ISO 8.3.2

```
int j=12;
int &k=j;
  /* k ist Alias fuer j */
k++;          /* j == 13 */
```

Felder sind eine Sammlung von Objekten eines bestimmten Typs. Mit

ISO 8.3.4

```
int ia[17];
```

wird ein Feld der Dimension 17 vom Typ Integer (Menge der ganzen Zahlen) definiert. Die einzelnen Elemente werden mit 0 bis 16 adressiert (indiziert).

```
ia[5]=-8721;
```

Fernerhin ist es möglich, Speicherplatz für Objekte **dynamisch** zu erzeugen und freizugeben. Mit dem `new`–Operator wird ein Objekt vom spezifizierten Typ erzeugt und ein Zeiger auf dieses zurückgegeben. `delete` gibt den Speicherbereich wieder frei.

```
int* l=new int;
  /* neue Ganzzahl */
*l=44;
  /* weise Wert zu */
delete l;
  /* gebe Speicher frei */
```

Konstanten sind Objekte, die einmal initialisiert nicht mehr verändert werden können (nur Lesezugriff möglich). Konstanten müssen demnach bei ihrer Definition initialisiert werden.

ISO 3.9.3

```
const int x=22;
const int y;  /* Fehler */
```

Mit der Anweisung

```
const double *pc;
```

wird ein Zeiger `pc` auf ein konstantes Objekt vom Typ `double` definiert. Das Objekt darf nicht verändert werden; hingegen ist es möglich dem Zeiger ein anderes Objekt zuzuweisen.

```
*pc=3.14;     /* Fehler */
double d;
pc=&d;       /* erlaubt */
```

Aufgabe 5.1

Informieren Sie sich über die in diesem Abschnitt vorgestellten Konstrukte. Insbesondere verdeutlichen Sie sich den hier nicht weiter beschriebenen Zusammenhang zwischen Zeigern, Referenzen und Feldern.

a Welche syntaktische Form muß der Name für ein Objekt, Typ etc. haben ?

b Welche Standardtypen (das sind vorgebene, eingebaute Typen) stehen in C++ zur Verfügung ? Welche sind davon in C verfügbar ?

c Welches sind die gängigsten Operatoren in C++ ? Welche Präzedenzen haben diese ?

d Was macht der Operator `?:` ? Gibt es weitere ternäre Operatoren ? Welche ?

e Gibt es Operatoren auf Typen ? Welche ?

f Worin besteht der Unterschied zwischen 'a' und "a" ?

g Welche Auswirkungen haben die Schlüsselworte `extern`, `static`, `auto` und `register` auf die Deklaration ?

h Welchen Gültigkeitsbereich haben Namen ? Welche Funktion erfüllt in diesem Zusammenhang der Operator `::` ?

i Was sind Aufzählungstypen ?

j Was bedeutet ein `void*` Zeiger ?

k Wie werden mehrdimensionale Felder definiert und initialisiert ?

l Definieren Sie mit Hilfe des Schlüsselwortes `typedef` folgende neue Typen: konstante Ganzzahlen, Zeiger auf kurze Realzahlen, Zeiger auf konstante Zeichen, Felder mit 6 positiven langen Ganzzahlen.

m Welche Lebensdauer besitzen Objekte ? Was machen die Operatoren `new` und `delete` ?

5.2 Anweisungen

Wie auch in anderen prozeduralen Sprachen, stehen in C++ einige Sprachkonstrukte zur Verfügung, die die Ablaufsteuerung (Kontrollfluß) regeln. Im wesentlichen sind dieses **Ablaufverzweigungen** und **Schleifen**.

Fernerhin ist es in C möglich, daher auch in C++, mit dem **Sprungbefehl** `goto` zu vorher vereinbarten Marken zu springen. ISO 6.6

Im allgemeinen ist es nicht nötig und zeichnet nicht gerade eine Programmiererin bzw. einen Programmierer aus, solche Sprungbefehle zu benutzen. Daher sollten bei der Lösung der hier gestellten Aufgaben auf diese Möglichkeit verzichtet werden.

ISO 2.6 **Kommentare**, also Programmtexte, die zur Dokumentation dienen, und keinen Einfluß auf den Programmablauf haben, werden in C/C++ durch

```
/* Hier steht jetzt ein
      wichtiger Kommentar ! */
```

eingestreut. Ferner ist es möglich, durch `//` Kommentare einzuläuten. Der nachfolgende Text wird dann bis zum Zeilenende ignoriert.

Hinreichend kommentierte Programme zeichnen eine „gute“ Programmiererin bzw. einen „guten“ Programmierer aus und sind unabdingbar.

C++ erhebt den Anspruch der sogenannten **Software–Wiederverwendung** (re–use), d.h., bestimmte Programmteile werden zu späteren Zeitpunkten in anderen Zusammenhängen wiederverwendet („recycling“). Gerade in solchen Situationen ist es sinnvoll und notwendig, ausreichende und aussagefähige Kommentare in den Schnittstellen zu integrieren.

Aufgabe 5.2
Befassen Sie sich mit den Konstrukten zur Kontrolle des Programmablaufs.

a Welche Bedeutung hat die `break`–Anweisung beim `switch`–Befehl ?

b Wandeln Sie folgende `if` Anweisung in eine `switch` Anweisung um.

```
if (a==1)
 tag="Montag";
else if (a==2)
 tag="Dienstag";
else if (a==3)
 tag="Mittwoch";
else if (a==4)
 tag="Donnerstag";
else if (a==5)
 tag="Freitag";
else if (a==6)
 tag="Samstag";
else if (a==7 || a==0)
 tag="Sonntag";
else
 tag=(char*)0;
```

c Worin besteht der Unterschied zwischen einer `do while`– und einer `while`–Schleife ?

d Was bedeutet `for(;;);` ?

e Kommentieren Sie Ihre Programmtexte in den weiteren Aufgaben sinnvoll (soll heißen, daß im wesentlichen die zugrundeliegenden Ideen, Konzepte sowie Parameter in Funktionen dokumentiert werden sollten, hingegen nicht z.B. die Anweisung `i++;`).

Achten Sie darauf, daß Sie Kommentare nicht schachteln, da dieses i.allg. nicht erlaubt ist, d.h. der Übersetzer einen Fehler meldet.

5.3 Funktionen

Ein häufig verwendetes Sprachkonstrukt in C++
ISO 8.3.5 (gilt natürlich auch für C) ist die **Funktion**. Diese besteht aus vier Teilen: dem Typ des Rückgabewertes (oder `void`), dem Funktionsnamen, der Argument– bzw. Parameterliste und aus dem Funktionsrumpf. Die ersten drei Elemente werden auch als **Funktionsprototyp** bezeichnet.
Bevor Funktionen benutzt werden können, müssen diese dem Übersetzer mitgeteilt, also deklariert werden. Dieses geschieht durch die Angabe des Funktionsprototyps.
Funktionen, die keinen Rückgabewert haben, werden manchmal auch als **Prozedur**en bezeichnet.

```
int add(int a, int b) {
 return a+b;
}
```

Diese Funktion hat zwei **formale Parameter** `a` und `b`. Beim Aufruf der Funktion werden diese
ISO 5.2.2 durch die **aktuellen Parameter** ersetzt, d.h., jeder formale Parameter wird mit dem jeweiligen aktuellen Parameter initialisiert. Obige Parameterübergabe wird auch als **call by value** bezeichnet.
Dieses Verfahren hat den Nachteil, daß die jeweiligen Objekte kopiert werden; dieses kann gerade bei größeren Objekten zuviel Rechenzeit und/oder Speicherkapazität beanspruchen[1].
Eine Alternative hierzu bietet die sogenannte Übergabe von Parametern mittels **call by reference**. Hierzu wird das übergebene Objekt nicht kopiert, sondern nur ein alternativer Name (Referenz) vereinbart.

```
int inc(int& a) {
 return a++;
}
```

Der Adreßoperator `&` definiert `a` als Referenz auf eine Ganzzahl. Da hier keine lokale Kopie angefertigt wird, betreffen alle Änderungen des formalen Parameters natürlich auch den aktuellen Parameter.

```
int b=42;
int c=inc(b);  /* b==43 */
```

Referenzen bei der Parameterübergabe sollten demnach nur dann benutzt werden, wenn keine Nebeneffekte zu erwarten bzw. diese auch erwünscht sind.
Die Deklaration des formalen Parameters als konstante Referenz ermöglicht es anzuzeigen, daß der Parameter nur aus Effizienzgründen als Referenz deklariert wurde und keine Nebeneffekte zu erwarten sind. Diese Möglichkeit ist besonders dann von Vorteil, wenn die Funktion auch von weiteren Personen benutzt wird, die nicht die interne Struktur der Funktion kennen.

```
int dec(const int& a) {
 b=a-1;         /* erlaubt */
 a++;  /* Fehler, konstant */
 return b;
}
```

Die Deklaration mit `const` sollte immer benutzt werden, wenn dieses möglich ist.

Aufgabe 5.3
Informieren Sie sich, wie Funktionen in C++ deklariert bzw. definiert werden.

a Was sind Funktionspointer ?

b Was sind `inline`–Funktionen ?

[1] Dieser Nachteil läßt sich nicht vermeiden, wenn der aktuelle Parameter nicht verändert werden darf, d.h., die Veränderungen sind lokal.

c Implementieren Sie eine Funktion, die einen Teil aus einer Zeichenkette herauskopiert. Als Parameter sollen linker und rechter Index, die den Bereich der zu kopierenden Teilzeichenkette begrenzen, übergeben werden.

d Schreiben Sie ein C–Programm, welches einen Namen *name* von der Kommandozeile als Argument einliest, und dann `Moin, Moin` *name* auf den Bildschirm ausgibt. Benutzen Sie als Ausgabe die Funktion `printf` aus `stdio.h`.
Modifizieren Sie Ihr Programm dahingehend, daß auch eine beliebige Anzahl von Namen in der Kommandozeile übergeben werden kann.

e Wie ist es möglich, eine beliebige endliche Anzahl von Argumenten einer Funktion zu übergeben ?

f Referenzen bei der Parameterübergabe sind eingeführt worden, um unnötige Kopiervorgänge zu vermeiden. Gibt es eine weitere Möglichkeit, Objekte Funktionen ohne zusätzlichen Kopieraufwand zu übergeben ?

5.4 Klassen

ISO 9 Das **Klassenkonzept** ist in C++ von fundamentaler Bedeutung und soll nachfolgend ein wenig ausführlicher besprochen werden.
Die Einführung von Klassen ist eine Fortführung des von anderen Programmiersprachen, z.B. Modula–2, unterstützten Prinzips der **Datenabstraktion**.
Anhand eines einführenden Beispiels soll das Klassenkonzept verdeutlicht werden.

Oft ist es wünschenswert, einen Booleschen Datentyp zu besitzen. In älteren Version von C++ ist dieser jedoch nicht definiert. Erst die neuste Version enthält einen `bool` Datentyp. Anhand des folgenden Beispiels soll gezeigt werden, wie ein fehlender Boolescher Datentyp realisiert werden kann.

```
#define boolean int
#define true 1
#define false 0
```

Die Idee ist, den Booleschen Datentyp mit dem Ganzzahlentyp `int` gleichzusetzen. Hierzu werden sogenannte Makros mit `#define` definiert. Beim Übersetzungsvorgang ersetzt der Präprozessor nun die Zeichenkette `boolean` durch `int`, `true` durch `1` und `false` durch `0`.

In C++ besteht die Möglichkeit, Konstanten zu definieren. Durch nachstehende Anweisungen wird derselbe Effekt wie durch die Makros erzielt.

```
typedef int boolean;
const boolean true=1;
const boolean false=0;
```

Ein Problem tritt auf, wenn diese in folgenden Anweisungen benutzt werden sollen.

```
if (isupper(c)==true)
 /* tue etwas */
```

Die Funktion `isupper` ist eine Funktion aus dem C Standard, die übernommen wurde, um die Kompatibilität zwischen C und C++ zu gewährleisten. `isupper` liefert „wahr" zurück, wenn `c` ein Großbuchstabe ist. Da in C „wahr" einen von 0 verschiedenen Wert bedeutet (Konvention), ist nicht garantiert, daß dieser auch 1 ist. Daher wird i.allg. nicht verzweigt, auch wenn `c` ein Großbuchstabe ist.
Eine Möglichkeit wäre, die Funktionen neu zu implementieren, so daß gewährleistet wird, immer `0` oder `1` zu erhalten. Der Aufwand scheint aber in Anbetracht der Anzahl von Funktionen nicht angemessen und ist, wie sich gleich herausstellt, nicht nötig.

Es stellt sich die Frage, warum nicht einen neuen Datentyp definieren, der sich wie Boolescher Typ verhält, und mit bereits vorhandenen Funktionen zusammenarbeitet ?

Das in C++ unterstützte Klassenkonzept ermöglicht es, benutzereigene Datentypen komfortabel zu definieren. Anhand dieses einfachen Beispiels lassen sich schon die wesentlichen Gedanken veranschaulichen.

```
class boolean {
  /* privater Bereich */

  int value;
  boolean isEqual(int) const;

 public:
  /* oeffentlicher Bereich */

  boolean();
  boolean(int);
  boolean(const boolean&);

  boolean& operator=(const
                   boolean&);

  void set(const boolean&);
  void set(int);

  operator const char*() const;
  operator int() const;

  void print(ostream& =cout) const;
  boolean operator! () const;

  boolean
   operator==(const boolean&) const;

  friend boolean
   operator==(int,const boolean&);

  friend boolean
   operator==(const boolean&,int);

};
```

Die Klasse `boolean` enthält als einzige Datenkomponente eine Variable `value` vom Typ Ganzzahl. Folgendes wird vereinbart: ist `value 1`, entspricht dieses dem Wahrheitswert „wahr“; ist `value 0`, dann soll der Wahrheitswert „falsch“ sein. Des weiteren sind eine Reihe von Methoden und Operatoren definiert, die die Funktionalität von Booleschen Typen bestimmen (hier ist nur eine kleine Auswahl sämtlich denkbarer Funktionen implementiert).

Die Klassendefinition gliedert sich in zwei Bereiche, die den Zugriff auf die **Methoden** und **Attribute** regeln. Methoden sind klassenspezifische Funktionen (funktionale Komponente, member functions); Attribute sind klassenspezifische Variablen (Datenkomponente). ISO 9.4
Der Punktoperator `.` ermöglicht hierbei den Zugriff auf Komponenten. Auf Methoden und Attribute im **privaten Bereich** darf nur die Klasse selber zugreifen, d.h. der folgende Aufruf ist mit der obigen Implementierung nicht erlaubt, und der Übersetzer meldet einen Fehler.

```
boolean x;
x.value=17;    /* Fehler */
```

Die für den Benutzer verfügbaren Methoden sind im **öffentlichen Bereich** aufgeführt, der durch das Schlüsselwort `public` eingeleitet wird. Diese Trennung zwischen „sichtbaren“ und „unsichtbaren“ Elementen wird oftmals auch als **Kapselung** bezeichnet. Der Benutzer „sieht“ nur die funktionale Komponente, die interne Struktur, d.h., die Kodierung von Wahrheitswerten durch Ganzzahlen bleibt ihm „verborgen“. ISO 11.1
Natürlich ist es auch möglich, klassenspezifische Variablen im öffentlichen Bereich zu deklarieren, jedoch ist, wie dieses Beispiel zeigt, der Zugang auf Attribute durch den Aufruf von Methoden „sicherer“. Der Vorteil des Zugriffs durch Methoden ist es, daß fälschliche Anweisungen verhindert werden können (im Sinne von logisch falsch, syntaktische Fehler werden durch den Übersetzer festgestellt). Wäre z.B. obige Anweisung möglich, erhielte `x.value` den Wert `17`. Wie oben vereinbart sollten die Wahrheitswerte jedoch nur durch

`0` oder `1` ausgedrückt werden. Das weitere Verhalten wäre undefiniert.
Das Setzen von Wahrheitswerten geschieht durch die Methoden `set`. Die Implementierung hat folgende Gestalt.

```
void boolean::set(int val) {
 value=(val ? 1 : 0);
}

void boolean::set(const
 boolean& b) {
  value=b.value;
}
```

Die erste Methode setzt automatisch in Abhängigkeit vom Parameter `val` das Attribut `value` auf einen der möglichen Wahrheitswerte. Undefinierte Zustände wie oben werden so vermieden.
Ein weiteres Konzept wird bei diesen zwei Methoden sichtbar: das **Überladen von Funktionen**. Es sind zwei Methoden gleichen Namens definiert, die sich nur durch den Typ im Parameter unterscheiden. Hierdurch ist es möglich, gleiche Aufgaben (gleiche Intention, nämlich das Setzen von Wahrheitswerten) auch gleich zu benennen. Grundsätzlich wäre es auch möglich gewesen,

ISO 13

```
void boolean::set_int(int
 val) {
  // tue etwas
}

void boolean::set_bool(const
 boolean& b) {
  // tue etwas
}
```

zu definieren.
Funktionen zu überladen, ist eine häufig benutzte Technik der OO–Programmierung (OOP). Sie vereinfacht die Lesbarkeit und läßt Programme transparenter erscheinen.
Neben Funktionen können auch **Operatoren überladen** werden. Dieses führt zu einer weiteren Strukturierung und Erhöhung der Lesbarkeit. Ein Operator ist eine Funktion, deren Schreibweise unter Benutzung eines Operationszeichen verkürzt werden kann.

ISO 13.5

```
int a,b=3,c=4;
a=operator+(b,c);
 /* explizit */
a=b+c;
 /* Kurzform */
```

Für die vorgegebenen, eingebauten Typen existieren bereits eine Reihe von Operatoren, die auch überladen sind.
Allerdings ist es nicht möglich, neue Operationszeichen wie z.B. `#` oder `@` zu definieren, sondern nur bereits vorhandene Zeichen wie z.B. `!`, `*` oder `+`. Hieraus ergibt sich, daß Operationszeichen nur dann überladen werden sollten, wenn die ursprüngliche Bedeutung erhalten bleibt oder durch das Erscheinungsbild des Zeichens eine Bedeutung offensichtlich wird. Grundsätzlich ist es möglich, daß +–Zeichen für Boolesche Objekte neu zu definieren. Aber welche Semantik hat die Addition von Booleschen Variablen ? Sinnvoller ist es indes, den logischen Verknüpfungsoperator `&&` neu zu definieren, da die Semantik unmittelbar auf der Hand liegt.

```
boolean boolean::operator!()
 const {
  return boolean(!value);
}

boolean& boolean::operator=(
 const boolean& b) {
  value=b.value;
  return *this;
}

boolean boolean::operator==(
 const boolean& b) const {
  return
   boolean(value==b.value);
}
```

Im einzelnen sind der unäre Operator `!` (Negation), der Zuweisungsoperator = und der Ver-

gleichsoperator == überladen worden. Die Bedeutung ergibt sich trivialerweise.
Fernerhin ist es möglich, die Operatoren zur **Typkonversion** (Typumwandlungen) zu überladen. Die Syntax zur Typumwandlung ist i.allg.
ISO 12.3 durch (Typ) gegben, z.B.

```
double x=3.2;
int y=(int)x;
```

Hier wird eine Realzahl in eine Ganzzahl umgewandelt. Durch

```
boolean::operator int()
 const {
  return value;
}
```

wird ein Wahrheitswert in eine Ganzzahl umgewandelt.
Der Operator (Typumwandlung)`const char*` gibt den Wahrheitswert als String aus.

```
boolean::operator const char*
 const {
  return
   (value? "true" : "false");
}
```

Fernerhin existieren drei überladene Funktionen, die mit dem Namen der Klasse `boolean` bezeichnet sind. Diese Methoden werden auch **Konstruktoren** genannt, da sie bei der Instanzierung der Klasse aufgerufen werden. Hierdurch ist es möglich, den Variablen Vorgabewerte zu geben (Initialisierung).

```
boolean::boolean() {
 value=1;
}

boolean::boolean(int i) {
 set(i);
}

boolean::boolean(const
 boolean& b) {
  value=b.value;
}
```

Instanzen der Klasse `boolean` können mit drei verschiedenen Aufrufen erhalten werden.

```
boolean a;
 /* Vorgabewert wahr */
boolean b(0);
 /* Init. mit falsch */
boolean c(b);
 /* Init. mit c=b    */
```

Des weiteren existieren zwei überladene Operatoren, die nicht zur eigentlichen Klassendefinition gehören, sondern mit dem Schlüsselwort `friend` als **befreundete Funktionen** deklariert sind. ISO 11.4

```
boolean operator==(int i,
 const boolean& b){
  return b.isEqual(i);
}

boolean operator==(const
 boolean& b, int i){
  return b.isEqual(i);
}
```

Befreundeten Funktionen ist es erlaubt, auf den privaten Bereich der Definition zuzugreifen, d.h., der Zugriffsmechanismus wird hierdurch explizit umgangen. Die befreundeten Funktionen rufen die private Methode `isEqual` auf, die vergleicht, ob eine Ganzzahl als Boolescher Wert interpretiert mit `value` identisch ist.

```
boolean boolean::isEqual(
 int i) const {
  if (value && i>=1)
   return boolean(1)
  else
   return boolean(0);
}
```

Mit Hilfe dieser Funktionen ist nun folgende Anweisung möglich.

```
const boolean true(1);
if (isupper(c)==true)
  /* tue etwas */
```

Die explizite Schreibweise lautet dementsprechend wie folgt:

```
const boolean true(1);
if (operator==(isupper(c),
               true))
  /* tue etwas */
```

Aufgabe 5.4

Informieren Sie sich ausführlich über die in diesem Abschnitt vorgestellten Begriffe wie Klasse, Zugriffsmechanismus, Überladung, Typkonversion und befreundete Funktionen sowie über deren Syntax in C++.

a Erklären Sie die Bedeutung von `const` als Postfix von Methodendeklarationen.

b Betrachten Sie die folgende Prozedur.

```
const char* boolean::error()
 const {
  /* tue etwas */
  return "error";
}
```

Welche Bedeutung hat `const` hier als Präfix und als Postfix ?

c Warum wurde der Operator == dreimal definiert ?

d Erläutern Sie den Unterschied zwischen Objekten, die mit `class` bzw. mit `struct` definiert sind.

e Erweitern Sie den Booleschen Typ um weitere logische Verknüpfungsoperatoren (und, oder, exklusiv–oder, nicht–und). Welche Operationszeichen erscheinen hier sinnvoll ?

f Implementieren Sie einen komplexen Zahlen–Typ `complex`.

g Implementieren Sie einen Typ zur Angabe von Winkelgrößen. Welche Operationen sind auf Winkeln sinnvoll ? Welche Operationszeichen lassen sich verwenden ?

5.5 Eingabe und Ausgabe

In C++ sind keine eigenen Schlüsselwörter für die Eingabe bzw. Ausgabe vorgesehen. Diese lassen sich viel mehr durch die vorhandenen Konstrukte ausdrücken.

Die zugrundeliegende Idee ist es, Sequenzen von Zeichen, die sogenannten **Ströme** (streams), zu modellieren. Die Zeichenfolgen können dann zum Beispiel auf dem Bildschirm sichtbar gemacht oder über die Tastatur eingegeben werden.

Mit Hilfe von auf Strömen definierten Operatoren und Methoden ist es möglich, Objekte in eine zeichenorientierte Darstellung umzuwandeln.

Für die Standardtypen sind hierzu die **Verschiebeoperatoren**[2] (shift) überladen worden.

```
>> x
```

Dieses bedeutet, daß etwas in das Objekt `x` hineinfließt, während

```
<< x
```

das genaue Gegenteil bedeutet.

In der Datei `iostream.h` sind einige Ströme definiert, die die Standardeingabe (Tastatur) und die Standardausgabe (Bildschirm) betreffen.

```
int x=1204;
cout << x << flush;
```

Mit diesen Anweisungen wird eine Variable vom Typ Ganzzahl deklariert, der Wert `1204` zugewiesen und auf dem Bildschirm ausgegeben.

Der Operator << übergibt das Objekt `x` dem Standardausgabestrom `cout`. Neben Objekten,

[2] In UNIX™ bietet i.allg. die Shell die Möglichkeit, die Eingabe bzw. Ausgabe umzuleiten. > bedeutet, die Ausgabe in die nachfolgend angegebene Datei zu schreiben. < liest aus einer Datei. Die Bedeutung ist also ein wenig anders als in C++.

die dem Strom übergeben werden sollen, können sogenannte **Manipulatoren** benutzt werden, um den Strom zu steuern.
In der Datei `iomanip.h` ist u.a. der Manipulator `flush` definiert, der dafür sorgt, daß das übergebene Objekt auch tatsächlich ausgegeben wird, da dem Ausgabestrom `cout` übergebene Objekte bzw. deren Zeichendarstellung zuerst in einem Puffer abgelegt werden.

Aufgabe 5.5
Informieren Sie sich über die gängigen Manipulatoren zur Ausgabe.

- **a** Wie ist es möglich, Ganzzahlen dezimal, oktal bzw. hexadezimal auszugeben ?
- **b** Wie lauten die Anweisungen, unter Benutzung der parametrischen Manipulatoren `setw` und `setfill` die Sequenz `*********!` auf dem Bildschirm auszugeben (ohne Anwendung von expliziten Schleifenkonstrukten) ?
- **c** Welche unterschiedlichen Auswirkungen haben `endl` und `'\n'` ?
- **d** Wie kann der Bildschirm (bzw. ein Fenster bei Bedienoberflächen) gelöscht werden ? (Trick !)

Der vordefinierte Strom für die Eingabe von der Tastatur ist mit `cin` bezeichnet. Um eine Variable z.B. mit einem Wert zu belegen, ist folgende Befehlssequenz möglich:

```
int x;
cout << "Geben Sie "
        "eine Zahl ein : "
     << flush;
cin >> x;
```

Aufgabe 5.6
Informieren Sie sich über die gängigen Manipulatoren zur Eingabe.

- **a** Welche Auswirkungen hat der Manipulator `ws` auf den Eingabestrom ?
- **b** Was bedeuten die folgenden Anweisungen (Funktionsweise) ?

```
int x,y;
cin >> x >> y;
```

Die Klasse `ios` ist die Basisklasse der verschiedenen Ein– und Ausgabestromklassen. In dieser Klasse sind verschiedene Methoden und Konstanten definiert, die zur Kontrolle des Datenstroms dienen.
Ähnlich wie bei den oben genannten Manipulatoren kann zum einen die Ein– bzw. die Ausgabe formatiert werden. Zum anderen ist es möglich, den Status des Datenstroms abzufragen. Diese ist besonders dann von Vorteil, wenn nicht die Standardein– bzw. –ausgabe benutzt wird.

Aufgabe 5.7
Informieren Sie sich über die Möglichkeiten zur Abfrage des Status eines Datenstroms.

- **a** Durch welche Methode ist es möglich zu überprüfen, ob bei der Eingabe ein Fehler gemacht wurde ?
- **b** Wie lautet die Ausgabe nach Ausführung der folgenden Befehlssequenz ?

```
cout.width(6);
cout.fill('*');
cout << '[' << 12
     << ']' << endl
     << '[' << 12
     << ']' << flush;
```

- **c** Welche Unterschiede im Vergleich zu **b** existieren zur Ausgabe der nachfolgenden Anweisungen ?

```
cout << setw(6)
     << setfill('*')
     << '[' << 12
     << ']' << endl
     << '[' << 12
     << ']' << flush;
```

Selbstverständlich ist es auch möglich, aus Dateien zu lesen bzw. in diese zu schreiben. In der Datei `fstream.h` sind Klassen zu Dateibearbeitung deklariert.

```
ofstream dat("xyz",ios::trunc|
             ios::out);
dat << "C++ macht Spass"
    << endl;
```

Durch diese Anweisungen wird eine Datei mit dem Namen `xyz` angelegt, falls diese noch nicht existiert, bzw. der Inhalt einer Datei mit gleichem Namen gelöscht. Anschließend wird `C++ macht Spass` in diese geschrieben.

Aufgabe 5.8
Die prinzipielle Vorgehensweise bei der Dateibearbeitung ist nicht anders als bei der Standardein– bzw. –ausgabe. Informieren Sie sich, wie der Schreib– bzw. Lesemodus bestimmt werden kann.

a Wie kann es erreicht werden, neue Daten am Ende einer bestehenden Datei anzuhängen ?

b Wo ist der Unterschied zwischen `ios::trunc||ios::out` und `ios::trunc|ios::out` ?

Für die Standardtypen in C++ sind bereits die Operatoren << und >> überladen worden. Praktisch wäre es, wenn es eine Möglichkeit gäbe, benutzerdefinierte Klassen Ein– oder Ausgabeströmen zu übergeben.
Dieses soll an der in Abschnitt 5.4 eingeführten Klasse `bool` verdeutlicht werden. Die Klassendefinition wird um die folgende befreundete und überladene Funktion ergänzt.

```
class boolean {
  /* Definition wie bisher */
  friend ostream& operator<<(
   ostream&, const boolean&);
};
```

Die Implementierung hat folgende Gestalt

```
ostream& operator<<(ostream& o,
 const boolean& b) {
  return
   (o << (const char*) b);
}
```

Der Wahrheitswert des Objektes des formalen Parameters `b` wird durch die Aufruf des Operators `const char*` als Zeichenkette ausgegeben und dem Ausgabestrom `o` übergeben. Die Anweisungen

```
const boolean true(5);
cout << true << flush;
```

geben auf dem Bildschirm `true` aus.

Der aufmerksamen Programmiererin bzw. dem aufmerksamen Programmierer wird aufgefallen sein, daß auch ohne die zusätzliche Definition der befreundeten Funktion bzw. des Operators die obigen Anweisungen eine Ausgabe auf dem Bildschirm erzeugen.
Da der Operator < < für Boolesche Objekte nicht definiert ist, versucht der Übersetzer eine Typumwandlung vorzunehmen. Da der Typkonversionsoperator `(int)` definiert ist, der ein Objekt der Klasse `boolean` in ein Ganzzahlobjekt umwandelt, wird dieser bei der Ausgabe implizit aufgerufen, d.h. die explizite Ausführung der Anweisungen lautet:

```
cout << (int) true << flush;
```

Auf dem Bildschirm wird dementsprechend `1` ausgegeben.

Die bisher gemachten Überlegungen sollen anhand einer praktischen Aufgabe erprobt und vertieft werden.

Aufgabe 5.9
Oftmals (z.B. bei Abwesenheit des Anwenders bzw. der Anwenderin) kann es von Vorteil sein, die Eingabe nicht direkt von der Tastatur sondern aus einer Datei einzulesen. Wünschenswert ist es, durch Angabe von Schaltereinstellungen beim Programmstart zwischen Eingabe aus einer Datei und interaktiver Eingabe zu wählen [3]. Ähnliches gilt selbstverständlich auch für die Ausgabe.

a Wie ist es möglich, Parameter beim Programmaufruf zu übergeben ?

b Implementieren Sie in C++ folgende Spezifikation:
Es soll eine zweidimensionale Matrix (Ganzzahlen) eingelesen und dann die zugehörige Transponierte ausgegeben werden. Hierbei soll zwischen Eingabe aus einer Datei und interaktiver Eingabe sowie zwischen Ausgabe in eine Datei und Ausgabe auf dem Bildschirm zu Programmstart gewählt werden können. Die Namen der Dateien sollten frei wählbar sein.

Das Dateiformat sei wie folgt: in den ersten beiden Zeilen der Datei stehe die Anzahl der Zeilen bzw. der Spalten der Matrix, daraufhin folge dann zeilenweise die Matrix.
Beispiel:

```
  4
  5
 12    4  69   15    30
966  161  24   23     3
 16   10   1    6  2666
 23   -1   0  883   758
```

Es ist eine geeignete (rechtsbündige) Formatierung zu wählen. Gehen Sie davon aus, daß die Dimension der Matrix nicht größer als 10×10 ist.

c Nützlich ist es, wenn in der Datei Kommentare eingefügt werden können, die beim Einlesen nicht beachtet werden.

Erweitern Sie Ihre Datei–Einlese–Prozedur zum Überlesen von Kommentaren. Ein Kommentar wird durch ein # in der ersten, d.h. ganz linken, Spalte eingeleitet. Nachfolgende Zeichen bis zum Zeilenende sind dann der eigentliche Kommentartext.

[3] In UNIX™ OS/2™ und MS-DOS™ ist es möglich, die Eingabe und Ausgabe mittels < und > umzuleiten. Diese Möglichkeit soll aber in dieser Aufgabe außer acht gelassen werden.

Lektion 6

Konstruktoren und Destruktoren

ISO 12.1
ISO 12.4

Konstruktoren und **Destruktoren** sind spezielle Methoden einer Klasse (vgl. Lektion 5.4). Wie bereits der Name erahnen läßt, wird ein Konstruktor beim Erzeugen (Konstruktion) und ein Destruktor beim Zerstören (Destruktion) einer Instanz aufgerufen.
Oftmals ist es jedoch nicht offensichtlich, daß Instanzen erzeugt bzw. zerstört werden. Diese daraus hervorgerufenen (impliziten) Funktionsaufrufe können besonders bei dynamischen Datenstrukturen (Klassen) zu unvorhersehbaren und nicht reproduzierbaren Fehlern führen.
In dieser Lektion soll der Umgang mit diesen Funktionen geübt werden.

6.1 Konstruktoren

Konstruktoren sind spezielle Methoden einer Klasse, die bei der Instanziierung aufgerufen werden.

6.1.1 Allgemeines

Konstruktoren sind Funktionen, die keinen Rückgabewert (auch nicht `void` !) besitzen dürfen. Der Name des Konstruktors ist mit dem Namen der Klasse identisch.
Eine mögliche Implementierung für Ganzzahl–Objekte ist:

```
class integer {
  int   value;
 public :
   integer(int val) {
    value=val;
   }

};
```

Bei der Erzeugung einer Instanz der Klasse `integer` ist nun ein Parameter (Vorgabewert) zu übergeben.

```
integer ganzzahl(20);
```

Bei Aufruf dieser Anweisung wird eine Instanz der Klasse `integer` erzeugt. Hierzu wird der Konstruktor aufgerufen und die Komponente `value` mit `20` initialisiert.
Natürlich ist es auch möglich, mehrere Parameter zu übergeben oder weitere Anweisungen im Konstruktor auszuführen.
Eine vereinfachte Schreibweise ergibt sich für die Initialisierung von Komponenten einer Klasse. Durch Angabe im Kopf der Konstruktordefinition ist eine **Initialisierung** mit bestimmte Werten möglich.

```
class complex {
  int real,imag;
 public :
  complex(int r):
   real(r),imag(0) {}

  complex(int r,int i):
   real(r),imag(i){}
};
```

Die durch Kommata getrennte Aufführung der zu initialisierenden Elemente wird als
ISO 12.6.2 **Komponenteninitialisierungsliste** bezeichnet.
Die Ausführung eines Konstruktors gliedert sich in zwei Abschnitte:

1. Initialisierung über die Komponenteninitialisierungsliste

2. Anweisungen im eigentlichen Funktionsrumpf

Zwei Konstruktoren einer Klasse sind besonders hervorzuheben:

- der parameterlose **Standardkonstruktor** und

ISO 12.8
- der sogenannte **Kopierkonstruktor**

Die Deklaration des Kopierkonstruktors für eine Klasse `X` hat i.allg. die Gestalt `X::X(const X&)`, d.h. die neu zu erzeugende Instanz wird mit einer bereits existierenden initialisiert.
Fernerhin muß der **Zuweisungsoperator** = einer Klasse genau betrachtet werden, da dieser mit dem Kopierkonstruktor in engem Zusammenhang steht.
Zuerst soll aber geklärt werden, in welchen Situationen der Konstruktor überhaupt aufgerufen wird.

- Instanziierung eines Objekts der Speicherklasse `auto`. Das sind i.allg. lokale Instanzen innerhalb eines Blocks, z.B.

  ```
  integer x(19+94);
  ```

- Instanziierung globaler und statischer Objekte. Der Konstruktor wird beim Programmstart aufgerufen.

- Instanziierung von temporären Objekten. Dieses ist bei Funktionsparametern und –rückgabewerten sowie bei Typumwandlungen der Fall. Als Beispiel sei gegeben:

  ```
  void f(integer x) {
   /* tue etwas */
   return;
  }

  int main() {
    integer y;
    f(y);
  }
  ```

  Hier wird zum einen der parameterlose Standardkonstruktor für `y` aufgerufen; zum anderen der Kopierkonstruktor für `x`, der beim Aufruf von `f(y)` die Instanz `y` nach `x` kopiert.

- Instanziierung dynamischer Objekte, d.h. durch Aufruf mit dem `new` Operator.

  ```
  integer *y=new integer(12);
  ```

Des weiteren gelten mit dem Umgang von Konstruktoren und dem Zuweisungsoperator folgende Regeln:

1. Eine Klasse ohne expliziten Konstruktor erhält automatisch einen leeren parameterlosen Konstruktor.

2. Eine Klasse ohne expliziten Kopierkonstruktor erhält diesen automatisch (bitweise Kopie).

3. Eine Klasse ohne expliziten Zuweisungsoperator erhält diesen automatisch (bitweise Kopie).

4. Die ersten drei Regeln werden automatisch rekursiv auf Basisklassen und Datenkomponenten einer Klasse angewendet.

5. Wird in einem Konstruktor nicht explizit beschrieben, wie Basisklassen und Datenkomponenten initialisiert werden sollen, werden die Standardkonstruktoren automatisch angewendet.

Die letzten beiden Regeln betreffen insbesondere das Konzept der **Vererbung**, welches erst in der nächsten Lektion ausführlich beschrieben wird.
Die aufgeführten Überlegungen sollen nun anhand einiger Beispiele verdeutlicht werden. Wie schon ersichtlich wurde, werden einige Aktionen automatisch, d.h., vom Übersetzer, durchgeführt, die u.U. zu einem Fehlverhalten führen können.

```
class string {
  char* s;
 public:

// Methoden und Operatoren

 string();
 string(const char*);
~string();

};
```

In der aufgeführten Zeichenketten–Klasse `string` werden zwei Konstruktoren und ein Destruktor definiert.

```
string::string() : s(0) {}

string::string(const char* c):
 s(0){
  if (c) {
   s=new char[strlen(c)+1];
   strcpy(s,c);
  }
}

string::~string() {
 delete [] s;
}
```

Eine mögliche Benutzung dieser Klasse könnte sein:

```
string s1("Uni Hildesheim");
string s2("Institut BS&RV");
s1=s2;
```

Unter der Annahme, daß kein Zuweisungsoperator definiert wurde, wird automatisch eine elementweise (bitweise) Zuordnung ausgeführt, d.h., dem Zeiger `s1.s` wird `s2.s` zugewiesen. Beide Zeiger zeigen anschließend auf `Institut BS&RV`. Drei Probleme treten auf:

- Beide Zeiger deuten auf denselbem Speicherbereich. Eine Änderung in einer der Zeichenketten betrifft auch die andere.
- Der allokierte Speicherbereich für `Uni Hildesheim` wird nicht freigegeben.
- Der Speicherbereich für `Institut BS&RV` wird zweimal freigegeben.

Um diese Probleme zu verhindern, muß also ein neuer **Zuweisungsoperator** definiert werden.

```
string& string::operator=(
 const string& c) {
  delete s;
  if (c.s) {
   s=new char[strlen(c.s)+1];
   strcpy(s,c.s);
  }
  else
   s=(char*)0;
  return *this;
}
```

Ein bereits benutzter Speicherplatz wird wieder freigegeben und der von der Zeichenkette `c` allokierte Speicher dupliziert. Die drei o.g. Probleme treten nicht mehr auf.

Allerdings zeigt sich, daß dieses allein noch nicht ausreichend ist.

```
string s1("Uni Hildesheim");
string s2;
s2=s1;
```

Diese und die nächsten Anweisungen sehen auf den ersten Blick ziemlich identisch aus und man würde vermuten, daß diese auch die gleichen Aktionen ausführen.

```
\index{Universit{\"a}t Hildesheim}%
string s1("Uni Hildesheim");
string s2=s1;
```

Hier tritt jedoch eine weitere Besonderheit von C++ auf. Anstatt bei der Definition von Objekten einen ein–parametrischen Konstruktor aufzurufen, z.B.

```
string s1("Uni Hildesheim");
```

kann dieses gleichwertig durch Angabe mit = geschehen, also

```
string s1="Uni Hildesheim";
```

Hierbei wird nicht der Zuweisungsoperator angewendet, sondern der zugehörige Konstruktor `string::string(const char*)` aufgerufen. Bei den Anweisungen im ersteren Beispiel verhält es sich ähnlich. Im ersten Fall handelt es sich um eine **Zuweisung** (`s1` und `s2` sind zum Zeitpunkt der Anweisung mit = bereits definiert). Im zweiten Fall handelt es sich um eine **Initialisierung**, da `s2` zum Zeitpunkt der = Anweisung noch nicht existiert, d.h. ein entsprechender Kopierkonstruktor wird aufgerufen. Da jedoch keiner explizit definiert wurde, wird eine elementweise (bitweise) Kopie durchführt. Dieses führt wiederum zu den oben aufgeführten Problemen. Deshalb muß zu den bereits bestehenden Konstruktoren ein dritter definiert werden.

```
string::string(const
 string& c):s(0){
  if (c.s) {
   s=new char[strlen(c.s)+1];
   strcpy(s,c.s);
  }
}
```

Erst durch diese beiden Ergänzungen wird das Verhalten der Zeichenketten–Klasse korrekt.
Um Fehlverhalten durch implizite Konstruktoraufrufe zu vermeiden, sollte auf jeden Fall für eine Klasse `X`

- ein Zuweisungsoperator

  ```
  X& X::operator=(const X&)
  ```

- und ein Kopierkonstruktor

  ```
  X::X(const X&)
  ```

definiert werden.
Folgende Überlegung sollte gemacht werden, wenn einem nicht bewußt ist, ob es sich bei einer Anweisung um eine Zuweisung oder um keine handelt:

> Wird in der Anweisung ein neues Objekt konstruiert, also Speicherplatz alloziert, so wird der Konstruktor aufgerufen und keine Zuweisung durchgeführt.

Aufgabe 6.1
Verinnerlichen Sie sich den Sinn und die Aufrufsyntax von Konstruktoren. Verdeutlichen Sie sich den Unterschied zwischen Zuweisung und Initialisierung.

a Ein häufiger Fehler ist folgende Anweisung.

```
integer xyz();
```

Beabsichtigt ist, eine Instanz der Klasse `integer` zu erzeugen. Wieso wird hier nicht der parameterlose Konstruktor der Klasse `integer` aufgerufen ?

b Konstruktoren existieren nicht nur für selbstdefinierte Klassen, sondern auch für die in C/C++ vorgegebenen Typen. Was bedeutet und worin liegt der Unterschied zwischen

```
int *x=new int(5);
```

und

```
int *x=new int[5];
```

c Betrachten Sie die Klasse `person` mit

```
class person {
  string name;
  string address;
 public:

  person(const string&,
         const string&);

  person(const person&);
  operator=(const person&);
};
```

1. Erklären Sie den Unterschied einer Definition des Konstruktors durch

   ```
   person::person(const string&
    n,const string& a) {
     name=n;
     address=a;
   }
   ```

   bzw. durch

   ```
   person::person(const string&
    n,const string& a) :
     name(n), address(a) {}
   ```

2. Implementieren Sie den Zuweisungsoperator sowie den Kopierkonstruktor.

6.1.2 Funktionen

Beim Aufruf von Funktionen, sowohl bei freien als auch bei klassenspezifischen (Methoden), werden häufig Konstruktoren (implizit) aufgerufen. Hier befindet sich eine häufige Fehlerquelle, insbesondere für Anfängerinnen und Anfänger.
Das folgende Beispiel soll zeigen, an welchen Stellen der **Konstruktor** beim Aufruf von Funktionen aufgerufen wird. Im wesentlichen geschieht dieses bei der Definition der **Funktionsparameter** und des **Rückgabewert**es.

```
string do_something(string v)
 {

// tue etwas ...

 string x("Professor");
 return x;
}


int main() {
 string y("Student"),z;
 z=do_something(y);

 return 0;
}
```

Entgegen vielleicht ersten Überlegungen wird in den oben genannten Anweisungen insgesamt fünfmal ein Konstruktor aufgerufen, und zwar:

- `string::string(const char*)` für `y`.
- `string::string()` für `z`.
- `string::string(const string&)` für `v`.
- `string::string(const char*)` für `x`.
- `string::string(const string&)` für den Rückgabewert (temporäre Instanz)

Anschließend wird dann noch einmal der Zuweisungsoperator, der die temporäre Instanz `z` zuweist, aufgerufen. Schnell wird erkennbar, welche Bedeutung der im letzten Abschnitt beschriebene **Kopierkonstruktor** hat.

Als nächstes stellt sich die Frage, wie diese unnötigen Kopien vermieden werden können. Eine Möglichkeit, die bereits häufig benutzt wurde, ist, Funktionsparameter als **call by reference** zu deklarieren. Dieses geschieht in C++ mit dem `&`–Zeichen als Postfix.
Durch die Deklaration `const` als Konstante

wird anderen Anwenderinnen und Anwendern verdeutlicht, daß der entsprechende Parameter nicht verändert wird, sondern nur aus Effizienzgründen eine unnötige Kopie vermieden werden soll. Die Funktion könnte also ungefähr folgende Deklaration haben:

```
string do_something(const
 string v&)
```

Schwieriger hingegen ist es, den Kopiervorgang des Rückgabewertes zu verhindern. Auf jeden Fall sollte folgendes nicht gemacht werden:

```
string& do_something(const
 string& v){
 // wie gehabt ...
  string x;
 // tue etwas mit x

  return x;
 /* Speicherfehler !!! */
}
```

Hier wird zwar der Kopiervorgang vermieden, jedoch ist `x` eine lokale Instanz der Funktion `do_something`, d.h., nach Rücksprung aus der Funktion wird der benutzte Speicherplatz wieder freigegeben.
Wird `x` als Referenz in der aufrufenden Funktion (hier `main`) weiterhin sichtbar, kann dieses u.a. Fehler zur Laufzeit zur Folge haben. Solche fehlerhaften Konstrukte sind also strikt zu vermeiden.

Aufgabe 6.2
Verdeutlichen Sie sich das hier vorgestellte Beispiel.

a Wie ist es möglich, Kopiervorgänge bei der Rückgabe von Funktionswerten korrekt zu vermeiden ?

6.1.3 Typumwandlung

Konstruktoren können auch zur **Typumwandlung** benutzt werden. Wie bereits in den vorherigen Abschnitten soll dieser Zusammenhang anhand einiger Beispiele verdeutlicht werden.
ISO 12.3.1

```
string s1="Samelson";
```

Die Zeichenkette `s1` wird mit `Samelson` initialisiert, d.h., es wird der Konstruktor `string::string("Samelson")` aufgerufen (vgl. Abschnitt 6.1.1). Hingegen wird bei

```
string s2;
s2="Knuth";
```

zuerst der parameterlose Konstruktor für `s2` aufgerufen. In der anschließenden Anweisung wird dann eine temporäre Instanz durch `string::string("Knuth")` erzeugt, die daraufhin mit `string::operator=` `s2` zugewiesen wird, d.h., die Zeichenkette `Knuth` vom Typ `char*` wird durch Aufruf des Konstruktors der `string`–Klasse in solch einen Typ umgewandelt.
Wie diese beiden Beispiele zeigen, ist es übersichtlicher und effizienter, wenn Definition und Initialisierung in einem Schritt ohne Erzeugung einer temporären Instanz durchgeführt werden.
Generell läßt sich sagen, daß jeder Konstruktor, der mit einem einzigen Parameter aufgerufen wird, vom Übersetzer zur Typumwandlung benutzt wird[1].

Aufgabe 6.3
Beschreiben Sie die Typumwandlung in den folgenden Beispielen. Wo wird genau welcher Konstruktur der Klasse `string` aufgerufen ?

a
```
void func(const string& s) {
// tue etwas ...
}
```

[1] Im vorgeschlagenen Standard wird ein neues Schlüsselwort `explizit` eingeführt, mit dem die implizite Umwandlung, wie in den hier vorgestellten Beispielen, verhindert werden kann.

```
int main() {
 func("Zuse");

 return 0;
}
```

b
```
string funcky() {
// tue etwas ...
 return "Torvalds";
}
```

c Die `string`–Klasse wird um einen zusätzlichen Konstruktor erweitert.

```
string::string(int i):s(0){
 if (i>0) {
  s=new char[13];
  sprintf(s,"%d",i);
 }
}
```

Erläutern Sie die folgenden Anweisungen.

1. `string s3=7;`
2. `string s4;`
   `s4=7;`

6.1.4 Felder

Der Verwendung von Konstruktoren bei der Erzeugung von Feldern einer selbstdefinierten Klasse (denn nur dort ist es möglich, Konstruktoren zu definieren) sind gewisse Einschränkungen auferlegt.

ISO 5.3.4

Wird ein Feld einer Klasse instanziiert, so wird der parameterlose **Standardkonstruktor** für jedes Feldelement aufgerufen.

```
string arr[47];
string* dynarr=new string[42];
```

Ab der C++ Version 3.0, die dieser Fibel zugrunde liegt, ist es alternativ möglich, einen Konstruktor mit einer beliebigen Anzahl von Parametern, die jeweils einen **Vorgabewert** besitzen müssen, zu definieren, der dann anstelle des parameterlosen Konstruktors bei der Instanziierung des Feldes aufgerufen wird.

Wird ein Feld mit `new` allokiert, so muß bei der Speicherfreigabe angegeben werden, daß es sich hierbei um ein solches handelt.

```
delete [] dynarr;
```

Wird die `[]`–Angabe vergessen, so wird zwar der Speicherbereich, den das Feld belegt, freigegeben, jedoch nicht der Speicherbereich, den die Instanzen des Feldes allokiert haben .
Durch die vollständige Angabe wird für jedes Feldelement der entsprechende **Destruktor** (s.u.) aufgerufen, der dann für eine korrekte und vollständige Speicherfreigabe und weitere Aufräumarbeiten sorgen kann.

6.2 Destruktoren

Etwas unkomplizierter ist der Umgang mit **Destruktoren**. Die Lebensdauer von Instanzen ist i.allg. begrenzt, d.h., wird der **Gültigkeitsbereich** eines Objekts verlassen, wird der von ihm allokierte Speicherbereich wieder freigegeben.
In vielen Fällen ist es aber zudem nötig, Aufräumarbeiten zu verrichten. Hierzu gehört u.a. die Freigabe des Speichers, der von dem Objekt zur Laufzeit dynamisch angefordert wurde.
Die Syntax des Destruktors entspricht der des Klassennamens mit dem Tildezeichen ˜ als Präfix. Destruktoren haben keinen Rückgabewert und besitzen auch keine Parameter (auch keine mit Vorgabewerten). Fernerhin ist es auch nicht möglich, diese speziellen Methoden zu überladen.
Destruktoren werden i.allg. nur implizit aufgerufen und zwar in den folgenden Situationen:

- Die Gültigkeit einer Instanz der Speicherklasse `auto` ist auf dem Block, in der diese definiert wurde, begrenzt. Der Destruktor wird für diese Instanz am Ende des Blocks automatisch aufgerufen.

- Der Destruktor für statische und globale Objekte wird am Ende eines Programms aufgerufen.
- Instanzen, die mit `new` dynamisch zur Laufzeit erzeugt worden sind, können durch `delete` wieder freigegeben werden. Hierbei wird automatisch der zugehörige Destruktor der Instanz aufgerufen.

Der Destruktor sollte den Speicherplatz, den die Instanz selber dynamisch anfordert (das ist nicht der Speicherplatz, den sie im Speicher belegt !) wieder freigeben.
Im dem Beispiel aus Abschnitt 6.1.1 wird der Platz für die Zeichenkette durch `delete [] s` wieder freigeben.
Wie bereits erläutert, erhält jede (selbstdefinierte) Klasse automatisch einen leeren Destruktor, falls nicht anderweitig ein solcher definiert ist.

Aufgabe 6.4
Informieren Sie sich über Sinn und Syntax von Destruktoren.

a Wie ist es möglich, Destruktoren explizit aufzurufen ? Welchen Sinn könnte so etwas haben ?

b Implementieren Sie den Destruktor für die Klasse `person` aus Aufgabe 6.1.

c Betrachten Sie das Programm

```
#include <iostream.h>

int main() {
 cout << "Uni Hildesheim\n";
}
```

und ergänzen Sie dieses, so daß folgende Ausgabe

```
Programmieren in C++ an der
Uni Hildesheim
macht viel Spass.
```

getätigt wird, ohne jedoch `main()` zu verändern.

d Im Abschnitt 6.1.2 wurde gezeigt, an welchen Stellen der Konstruktor aufgerufen wurde. Beschreiben Sie, wo und zu welchen Zeitpunkten der Destruktor aufgerufen wird.

Lektion 7

Vererbung

ISO 10 Das Prinzip der **Vererbung** ermöglicht es, vorhandene Klassen zu erweitern, indem **abgeleitete Klassen** die Daten– und die funktionale Komponente übernehmen und durch weitere Komponenten ergänzt werden können. Vererbung ist eines der wesentlichen Charakteristika von objektorientierten Programmiersprachen.

7.1 Einfachvererbung

Die grundlegende Idee ist, Gemeinsamkeiten zwischen verschiedenen Klassen zu finden und diese bezüglich eines **Generalisierung–Spezialisierung**–Kontextes in Beziehung zu setzen. Die generalisierte Klasse wird als **Oberklasse** oder **Basisklasse** bezeichnet, hingegen die spezialisierte als **Unterklasse** oder **abgeleitete Klasse**.
Die spezialisierte Klasse hat den Umfang der gewählten Basisklasse, d.h., diese erbt die klassenspezifischen Variablen und Methoden und kann gegebenfalls um weitere Daten– oder funktionale Komponenten erweitert werden.
Vererbung ermöglicht es, auf vorhandene Klassen zurückzugreifen sowie diese (in gewissen Grenzen) zu modifizieren und zu erweitern, um sie so weiterzuverwenden (**Software–Wiederverwendung**, re–use).

Anhand eines kleinen Beispiels soll die entsprechende Syntax in C++ erläutert werden.
Es sollen hierbei Ganzzahl–Objekte (`integer`) sowie sogenannte bereichsbeschränkte Zahlen (`gauge`) betrachtet werden. Bereichsbeschränkte Zahlen lassen sich anschaulich als Markierung auf einer Meßlatte beschreiben. Der Bereich der Meßlatte ist beschränkt, d.h., die Markierung kann nur bestimmte Positionen annehmen.
Zuerst soll aber die Ganzzahlklasse aus Lektion 6.1.1 um einige Methoden ergänzt werden.

```
class integer {
  int value;
 public :

  integer(int val =0) {
   value=val;
  }

  void set(int);
  int  get() const;
};
```

Die beiden Methoden `set` und `get` dienen zur Modifikation und zum Lesen des aktuellen Wertes.

```
void integer::set(int x) {
 value=x;
}

int integer::get() const {
 return value;
}
```

Die Klasse der bereichbeschränkten Zahlen ist eine Spezialisierung der Ganzzahl–Klasse (Basisklasse). Ein Objekt des Typs `gauge` ist eine Ganzzahl (der Wertebereich der Basisklasse sei idealerweise unbeschränkt) aus einem festdefinierten Wertebereich. Eine Klassendefinition könnte wie folgt aussehen.

```
class gauge :
 public integer {

  int max;
  int min;
 public :

  gauge(int,int,int);
  void set(int);
};
```

Die Methodendefinition lautet

```
gauge::gauge(int v,int l,
 int u) {
  max=u;
  min=l;
  set(v);
}

void gauge::set(int val) {
 value=(val>max ?
  max:(val<min ? min:val));
}
```

Der Konstruktor setzt die übergebenen Schranken für den Wertebereich und anschließend den eigentlichen Wert `value` durch Aufruf von `gauge::set`. Zwei Punkte seien hier hervorgehoben. Zum einen beinhaltet die Klasse `gauge` eine klassenspezifische Variable `value`, die sie von der Basisklasse `integer` (spezifiziert mit `class gauge : public integer`) erbt. Zum anderen wird die Methode `gauge::set` neu definiert, d.h. die geerbte Methode `set` von `integer` wird durch die neue Definition **verdeckt** bzw. **überschrieben**. Die neue Setz–Methode überprüft, ob der neue Wert `val` den Wertebereich verläßt und setzt dementsprechend `value`. Die Lese–Methode `get` braucht nicht verändert zu werden und kann deshalb einfach übernommen und benutzt werden.

```
gauge x(10,7,13);
cout << x.get()
     << endl; // integer::get
x.set(25);    // gauge::set
cout << x.get()
     << endl; // Ausgabe: 13
```

In dem angegeben Beispiel wird eine bereichsbeschränkte Zahl `x` mit dem Wertebereich [7,. . .,13] definiert und ihr der Wert 10 zugewiesen. Anschließend wird diese auf den maximalen Wert 13 gesetzt, da durch den übergebenen Parameter der Wertebereich verlassen wird.

Für Zeiger auf abgeleitete Klassen gilt folgende Regel: Ist eine Klasse `Y` von einer Klasse `X` abgeleitet (die Basisklasse muß dabei mit `public` spezifiziert worden sein), dann kann ein Zeiger auf eine Instanz vom Typ `Y` einem Zeiger vom Typ `X*` ohne explizite **Typkonversion** zugewiesen werden.

```
gauge* y=new gauge(10,7,13);
integer* x=y;
cout << x->get() << endl;
```

Selbstverständlich kann von abgeleiteten Klassen wiederum an andere Klassen vererbt und so eine **Klassenhierachie** aufgebaut werden.
Eine Oberklasse darf außerdem eine unbeschränkte Anzahl von Unterklassen haben. In diesem Abschnitt hat die Unterklasse jedoch nur (genau) eine Oberklasse (**Einfachvererbung**). Wie dieses erweitert werden kann, welches dann auf die **Mehrfachvererbung** führt, wird im nächsten Abschnitt dieser Lektion beschrieben.

Aufgabe 7.1
Informieren Sie sich über die Konstrukte der Vererbung in C++.

a In welcher Reihenfolge werden die Konstruktoren und Destruktoren der Ober– und Unterklasse aufgerufen ?

b Betrachten Sie dann folgende Klassendefinitionen.

```
class A {
 protected:
  int i;
 public :
  A();
  A(int);
 ~A();
  void f();
  void show();
};

class B : public A {
 public :
  B();
 ~B();
  void f();
};

class C : public B {
 public :
  C();
 ~C();
 void f();
};

A::A() : i(1) { cout << "A"; }
A::A(int j) : i(j) {
 cout << "A(" << i << ")";
}
A::~A() { cout << "~A"; }
void A::f() { cout << "A::f"; }
void A::show() {
  cout << "i=" << i; }

B::B() : A(0) {
 i=2; cout << "B"; }
B::~B() { cout << "~B"; }
void B::f() { cout << "B::f"; }

C::C() { i=3; cout << "C"; }
C::~C() { cout << "~C"; }
void C::f() { cout << "C::f"; }
```

Beschreiben Sie, welche Ausgaben auf dem Bildschirm (nachdem der Ausgabestrom mit `cout « flush;` geleert wurde) nach Auswertung der unten aufgeführten Anweisungen (Block) gedruckt werden.

1. `{ A x; x.show(); x.f(); }`
2. `{ B x; x.show(); x.f(); }`
3. `{ C x; x.show(); x.f(); }`
4. `{ A x=B(); x.show();`
   `x.f(); }`
5. `{ A* x=new C;`
   `x->show();`
   `x->f();`
   `delete x; }`

c Welche Auswirkungen auf den Zugriff der Basisklasse haben die Spezifizierer `private` und `public` ?

d Implementieren Sie eine Klasse `counter`. Ein Zähler besitzt die Eigenschaften einer bereichsbeschränkten Zahl mit folgenden Einschränkungen:

- Ein Zähler kann nur erhöht werden.
- Wird der Wertebereich verlassen, wird der Zähler auf den minimalen Wert gesetzt.

7.2 Mehrfachvererbung

Manchmal ist es sinnvoll, die Eigenschaft nicht von einer Basisklasse, sondern von mehreren zu erben. Auch diese Möglichkeit wird von C++ (im Gegensatz zu manch anderer OOPS) unterstützt.
Anstatt eine einzelne Basisklasse nach dem **:** in der Klassendeklaration anzugeben, kann eine Liste von Basisklassen, jede einzelne mit `public` oder `private` spezifiziert, folgen.

Probleme treten auf, wenn aufgrund der Mehrfachvererbung in der Klassenhierachie eine Basisklasse mehrfach auftritt, da hier Namenskonflikte zu erwarten sind.
Beispiel:

```
class A {
 /* Klassendeklaration */
 void f();
};

class B : public A {
 /* Klassendeklaration */
};

class C : public A {
 /* Klassendeklaration */
};

class D : public B, public C {
 /* Klassendeklaration */
};
```

Es stellt sich nun die Frage, wie auf die Methode `A::f` zugegriffen werden soll, da diese in `B` und in `C` aufgrund der Einfachvererbung vorhanden ist.

```
D d; d->f();
 /* Fehler, doppeldeutig */
```

Um diese Doppeldeutigkeit aufzulösen, muß die entsprechende Klasse angegeben werden; also entweder

```
d->B::f();
```

oder

```
d->C::f();
```

Mehrfachvererbung sollte auf jeden Fall sehr behutsam eingesetzt werden, da u.U. Mehrdeutigkeiten entstehen, die dann nur durch die explizite Angabe der Basisklasse aufgelöst werden können. Hierdurch verliert man die Transparenz, die durch die Vererbung eigentlich erzielt werden sollte. Die nachfolgende Lektion wird hierauf nochmals eingehen.

Aufgabe 7.2
Informieren Sie sich über die Mehrfachvererbung in C++.

a In welcher Reihenfolge werden die Konstruktoren und Destruktoren der Ober– und Unterklassen aufgerufen ?

Lektion 8

Virtuelle Methoden und abstrakte Basisklassen

Diese Lektion enthält einige weitere Konzepte, die sich in erster Linie an fortgeschrittene C++ Programmierende wenden.

8.1 Überladen von Funktionen

ISO 13 Zuerst soll das Konzept des **Überladen**s, welches bereits in Abschnitt 5.4 vorgestellt wurde, ein wenig vertieft werden.
Unter Überladen wird in der OOP die Möglichkeit bezeichnet, gleichnamige Funktionen (Methoden, Operatoren) innerhalb eines Gültigkeitsbereiches zu definieren, die sich nur in der Anzahl und/oder dem Typ der Parameter, d.h. des Funktionsprototypen, unterscheiden.
Als klassisches Beispiel sei die Funktion `abs` angeführt, die den Absolutbetrag einer Zahl (vom Typ `int, float, double`) liefert.

```
int abs(int n) {
 return n<0 ? -n: n;
}

float abs(float n) {
 return n<0 ? -n: n;
}

double abs(double n) {
 return n<0 ? -n: n;
}
```

Dieses Verfahren läßt sich auf Methoden, d.h. klassenspezifische Funktionen und auf Operatoren leicht übertragen.

An dieser Stelle sei auf den Unterschied zwischen **Überladen** und **Verdecken** bzw. **Überschreiben** gleichnamiger Methoden (Operatoren) bei abgeleiteten Klassen (vgl. vorherige Lektion) hingewiesen.
Dieses soll anhand der `integer`– und `gauge`–Klasse aus Abschnitt 7.1 beschrieben werden.
Die Methode `integer::set` dient zum Setzen einer Ganzzahl. Die dazu äquivalente Methode `gauge::set` aus der Klasse der bereichsbeschränkten Zahlen hat einen anderen Gültigkeitsbereich, da diese innerhalb der `gauge`–Klasse deklariert und definiert ist, d.h. die Methode wird nicht überladen, sondern überschrieben. Bei Aufruf in den Anweisungen

```
gauge x(10,7,13);
x.set(12);
```

wird die Setzmethode der `gauge`– und nicht der `integer`–Klasse ausgeführt. Diese kann jedoch weiterhin durch

```
x.integer::set(25);
```

explizit aufgerufen werden. Leider wird hierdurch die bereichsbeschränkte Zahl auf einen nichtdefinierten Wert gesetzt.

Der Unterschied zwischen Überladen und Überschreiben wird an folgenden Beispiel nochmals ersichtlich.

```
int y=76;

int main() {
 int y=19;
 cout << y << y+::y << endl;
}
```

Die globale Variable `y` wird in der Funktion `main` durch eine neuerliche Definition von `y` **verdeckt**, d.h., alle Anweisungen betreffend `y` werden auf diese lokale Variable angewendet. Durch die explizite Angabe von `::` ist es möglich, auf das globale Objekt zurückzugreifen. Allerdings ist es durch den `::` Operator nicht möglich, auf lokal definierte Objekte zuzugreifen.

Aufgabe 8.1
Machen Sie sich mit der Möglichkeit des Überladens vertraut. Beschreiben Sie den Unterschied zwischen Verdecken und Überladen.

- **a** Der `::`–Operator erlaubt es, auf verdeckte Methoden der öffentlichen Basisklasse zuzugreifen. Das o.g. Beispiel verdeutlicht jedoch die Probleme, die dadurch entstehen. Wie ist es möglich, diese expliziten Aufrufe nicht zu erlauben ?
- **b** Implementieren Sie eine freie Funktion `abs` für die `integer`– bzw. `complex`–Klasse aus Lektion 6.1.1.

8.2 Polymorphie

ISO 10.3 **Polymorphe** Methoden erlauben, abhängig vom spezifizierten Objekt, verschieden gestaltig zu wirken. Das nachfolgende Beispiel soll verdeutlichen, was dieses in der Praxis zu bedeuten hat.

```
integer* ptr=(integer*)0;
char a; int v;
cin >> a;
if (a=='i')
 ptr=new integer(0);
else
 ptr=new gauge(0,0,10);
cout << ptr->get() << endl;
cin >> v;
ptr->set(v);
```

Anhängig von der Benutzereingabe wird ein Zeiger `ptr` auf eine Instanz der Klasse `integer` bzw. `gauge` gesetzt. Das initialisierte Objekt wird ausgegeben und anschließend mit einem neuen Wert belegt.
Nicht offensichtlich ist, wo und warum diese Anweisungen zu Problemen führen.
`ptr` ist vom Typ Zeiger auf Ganzzahl. Wird jedoch diesem ein Objekt vom Typ `gauge` zugewiesen (was erlaubt ist, da `gauge` von `integer` abgeleitet ist) und dann die `set`–Methode aufgerufen, so wird folglicherweise die entsprechende Methode der `integer`–Klasse ausgeführt. Dieses ist jedoch nicht beabsichtigt, da die Setz–Methode in der abgeleiteten Klasse neu definiert wurde.
Das Problem ist also, daß ein Zeiger auf eine Basisklasse einer Instanz einer abgeleiteten Klasse zugewiesen wird. Wie ist nun möglich, dieses Problem ohne explizite **Typumwandlungen**, wie z.B. im folgenden Ansatz

```
if (a=='i')
 ptr->set(v);
else
 ((gauge*)ptr)->set(v);
```

geschehen, zu vermeiden ?
Polymorphe Methoden erlauben, abhängig vom eigentlichen Typ des referenzierten Objektes, die in dieser Situation „richtige“ Funktion anzuwenden.
In C++ werden solche Methoden in der Klassendeklaration durch den Präfix `virtual` spezi- ISO 10.3

fiziert. Die integer–Klasse würde dann wie folgt lauten:

```
class integer {
// wie gehabt ...

 virtual void set();

};
```

Dieser Sachverhalt wird des öfteren auch als **späte Bindung** bzw. Bindung zur Laufzeit bezeichnet, da in Abhängigkeit des Objektes die entsprechende Methode zur Laufzeit des Programmes ausgewählt wird, im Gegensatz zur **frühen Bindung**, d.h. einer Zuweisung der Methode zum Objekt zum Zeitpunkt der Übersetzung. Das Laufzeitsystem überprüft, von welchem Typ das Objekt ist, welches referenziert wird. Polymorhie ist also im Zusammenhang mit der Vererbung wichtig. Prinzipiell ist es möglich, alle Methoden als virtuell zu deklarieren, der Programmablauf wird jedoch dadurch verzögert. Einmal virtuell deklarierte Methoden sind auch in nachfolgend vererbten Klassen virtuell.

8.3 Abstrakte Basisklassen

ISO 10.4 Wie bereits der Titel dieses Abschnittes vermuten läßt, werden Klassen besprochen, die nicht konkretisiert, d.h. instanziiert, werden können. Bevor auf die entsprechende Syntax in C++ eingegangen wird, soll die Frage geklärt werden, welche Motivation hinter dieser Möglichkeit steht.
Im Prinzip ermöglichen abstrakte Klassen, ein Schema für die abgeleiteten Klassen vorzugeben, eine Art von Schnittstelle, ohne jedoch dabei auf Implementierungdetails einzugehen.
So könnte z.B. eine oberste Basisklasse topobject deklariert werden, die von den abgeleiteten Klassen verlangt, daß diese eine Methode debug zur Verfügung stellen, die es ermöglicht, Laufzeitinformationen über Instanzen zu erhalten.

```
class topobject {
 public:
  virtual void debug() const =0;
};
```

Da a-priori nicht feststeht, wie die Klassen in der Hierarchie aussehen, ist es schwer möglich, eine entsprechende Test–Methode debug vorzugeben. Statt dessen wird auf eine Implementierung vollständig verzichtet und nur Parameter und Rückgabewert spezifiziert.
Konkrete abgeleitete Klassen von topobject müssen nun eine Methode debug anbieten, die die beschriebene Funktionalität erfüllt.
Methoden, die zwar deklariert, aber nicht definiert werden, erhalten neben dem notwendigen Präfix virtual einen zusätzlichen Postfix =0. Solche klassenspezifischen Funktionen werden als **pure virtuelle Methoden** bezeichnet.
Für die integer– und gauge–Klasse könnte es z.B. folgendes bedeuten:

```
class integer :
 public topobject {

// wie gehabt ...

// zusaetzlich public
  virtual void debug()
   const;
};

class gauge :
 public integer {

// wie gehabt ...

//  zusaetzlich public
  virtual void debug()
   const;
};                    ,
```

mit den entsprechenden Definitionen

```
void integer::debug()
 const {
```

```
  cout << "integer "
       << value << endl;
}

void gauge::debug()
 const {
  cout << "gauge ["
       << min << ","
       << max << "] "
       << value << endl;
}
```

Obwohl keine Instanzen einer abstrakten Basisklasse existieren dürfen, ist es erlaubt, Zeiger auf solche Klassen zu definieren. Ein abschließendes Beispiel zeigt den Nutzen.

```
topobject* top=(topobject*)0;
integer i(5);
gauge   g(10,5,15);
top=&i;
top->debug(); // integer 5
top=&g;
top->debug();
        // gauge [5,15] 10
```

Aufgabe 8.2
Informieren Sie sich über die vielseitige Verwendung des Schlüsselwortes `virtual`.

a Was sind virtuelle Basisklassen ?

Lektion 9

Schablonen

ISO 14 In dieser Lektion wird gezeigt, wie sogenannte **Schablonen** (oder **Templates**) die oftmals hinderliche Typbindung in gewissen Grenzen aufheben.

9.1 Allgemeines

In Abschnitt 8.1 wurde gezeigt, wie durch **Überladen** in einem Gültigkeitsbereich gleiche Bezeichner mehrfach verwendet werden können. Eine nähere Betrachtung der verschiedenen Betragsfunktionen `abs` zeigt, daß diese in den wesentlichen Anweisungen übereinstimmen und sich nur in der Typbindung unterscheiden.
Durch die Einführung eines zusätzlichen Parameters, der den Typ des Parameters `n` der Betragsfunktion angibt, ist es möglich, die doppelte Kodierung zu vermeiden.
In C++ geschieht dieses durch folgende Definition:

```
template<class T> T abs(T n) {
 return n<0 ? -n : n;
}
```

Durch die Angabe `template<class T>` wird erklärt, daß `T` ein zusätzlicher Parameter ist, der den Typ des Objektes `n` und den des Rückgabewertes angibt.
Diese **Funktionsschablone** stellt also im Prinzip eine unendliche Menge von überladenen Betragsfunktionen zur Verfügung.
Damit diese mit verschiedenen Typen benutzt werden kann, muß ein Typ `T` die folgenden Operator und Methoden zur Verfügung stellen:

1. unärer Minusoperator − zum Invertieren
2. Vergleichsoperator <, um eine Instanz vom Typ `T` mit `0` vom Typ `int` in Beziehung zu setzen.
3. Kopierkonstruktor `T(const T&)`

Diese Schablone kann wie eine gewöhnliche Funktion aufgerufen werden.

```
int main() {
 int y=abs(-42);
}
```

Wird die `integer`–Klasse um die folgenden Methoden erweitert,

```
int integer::operator<(int c) {
 return value<c;
}

integer integer::operator-() {
 return integer(-value);
}
```

ist es möglich, die o.g. Schablone auch auf diesen Typ anzuwenden.

```
int main() {
 integer x(-42);
 integer y=abs(x);
}
```

Werden diese Methoden bzw. Operatoren nicht hinzugefügt, dann wird beim Übersetzungsvorgang eine Fehlermeldung ausgegeben.
Der Programmierende solcher Templatefunktionen hat also dafür zu sorgen, daß die im Funktionsrumpf vorkommenden, an den Schablonenparameter gebundenen Anweisungen (z.B. Methodenaufrufe) vollständig und korrekt vorhanden sind.
Hier zeigt sich die Notwendigkeit einer vernünftigen **Dokumentierung**, da besonders bei größeren Funktionen nicht vom Anwender bzw. von der Anwenderin verlangt werden kann, alle Einzelheiten und Abhängigkeiten nachzuvollziehen.

Das Konzept der Schablone läßt sich ohne weiteres auch auf die Definition von Klassen (Typen) erweitern. Anstatt z.B. einen Listentyp für Ganzzahlen `list_integer` und einen Typ für eine Liste `list_person` zu implementieren, kann durch Verwendung eines **Klassentemplate** einfach eine generische Liste `list_generic` mit einen zusätzlichen Typparameter definiert bzw. benutzt werden.
Die Syntax generischer Klassen soll anhand eines einfachen Beispiels nachfolgend kurz erläutert werden. Die Klasse `vector` soll einen einfachen Vektor–Datentyp realisieren.

```
template<class T> class vector {
  T*    elements;
  int   dim;

 public :

  vector(int =0);
  vector(const vector<T>&);
 ~vector();

  vector<T>& operator=(const
                    vector<T>&);
  T& operator[](int);

};
```

Folgende Definitionen sind noch notwendig. Man beachte besonders die im ersten Augenblick ungewöhnliche Syntax der Prototypen.

```
template<class T>
 vector<T>::vector(int d):
  dim(d), elements(new T[d]) {}

template<class T>
 vector<T>::vector(const
 vector<T>& v) {
 dim=v.dim;
 elements=new T[dim];
 for(int i=0;i<dim;i++)
  elements[i]=v.elements[i];
}

template<class T>
 vector<T>::~vector() {
  delete [] elements;
}

template<class T> vector<T>&
 vector<T>::operator=(const
 vector<T>& v) {
 if (this!=&v) {
  delete [] elements;
  dim=v.dim;
  elements=new T[dim];
  for(int i=0;i<dim;i++)
   elements[i]=v.elements[i];
 }
 return *this;
}

template<class T> T&
 vector<T>::operator[](int i) {
 if(i>=0 && i<dim)
  return elements[i];
 else
  exit(-1);
}
```

Der (Standard–) Konstruktor allokiert den notwendigen Speicher. Mit Hilfe des Kopierkonstruktors bzw. durch Anwendung des =–Operators ist es möglich, Felder zu initialisieren bzw. zuzuweisen. Der Indizierungsoperator `[]` erlaubt es letztendlich, auf einzelne Komponenten einer `vector`–Instanz zuzugreifen.

Aufgabe 9.1
Befassen Sie sich mit den Konzepten und Anwendungen, die durch die Einführung von Schablonen ermöglicht werden.

- **a** Implementieren Sie einen generischen Datentyp `array`, der es ermöglicht, typgleiche Elemente in einer zweidimensionalen Matrix abzulegen. Der Typ dieser Elemente sei frei wählbar, aber für alle Elemente konstant. Überladen Sie den `[]`–Operator.
- **b** Implementieren Sie eine generische Sortierfunktion `quicksort`. Welche Voraussetzungen müssen die zu sortierenden Elemente erfüllen ? Vergleichen Sie Ihre Implementierung mit der C–Funktion `qsort` aus `stdlib.h` bezüglich Benutzerfreundlichkeit und Wiederverwendbarkeit.
- **c** Warum ist folgender Funktionsprototyp in C++ nicht erlaubt ?

```
template<class T>
void nofun(double);
```

9.2 Standard Template Library

Die sogenannte **Standard Template Library** (STL) ist eine Klassenbibliothek, die in ähnlicher Form ebenfalls in **Ada** und **Scheme** vorhanden ist.
Diese Bibliothek für die Programmiersprache C++ wurde von **Alexander Stepanov** und **Meng Lee** implementiert (vgl. [SL94b]). Im Juli 1994 wurde diese Bibliothek, ähnlich wie die `stream`–Klassen, in den C++ Standard aufgenommen. Hieraus ergibt sich der Vorteil, daß alle zukünftigen Übersetzer, die dem Standard folgen, diese enthalten und so eine Portierung von Quelltexten ermöglicht wird.
Eine Einführung in die Konzepte und Programmierung ist in [Vil94] und [Str94b] zu finden.

Lektion 10

Fehlerbehandlung

ISO 15 In der Version 3.0 ist C++ um einige Mechanismen zur **Fehlerbehandlung** erweitert worden. Hierbei handelt es sich nicht um eine typische objektorientierte Erweiterung; diese fügt sich jedoch problemlos in die vorhandenen Konzepte ein.

Die zugrundeliegende Idee läßt sich wie folgt beschreiben: Fehler werden dadurch ausgelöst, daß eine sogenannte **exception** „geworfen" (`throw`) wird, die dann von einer Fehlerbehandlungsroutine (**exception handler**) „aufgefangen" (`catch`) und bearbeitet wird.

Wird z.Zt. noch nicht unterstützt !

Lektion 11

Projekt

In dieser letzten Lektion wird eine größere Aufgaben zu Analyse, Entwurf und Implementierung gestellt. Ziel soll es ein, die bisherigen Erfahrungen zu vertiefen.

11.1 Ein dynamisches System

Die Idee zur vorliegenden Aufgabe stammt von Dewdney aus [Dew89]. Ziel soll es sein, das soziale Verhalten von Gästen auf einer Party zu simulieren und zu studieren.
Um einen Überblick über den Verlauf der Party zu erhalten, wird eine Aufsicht für den Partyraum gewählt. Dieser wird in Zellen eingeteilt, d.h. er läßt sich als n $\times$ m Matrix beschreiben. Personen werden durch den Anfangsbuchstaben des Namens markiert. In der Mitte des Raumes befindet sich ein Tisch. Abbildung 11.1 verdeutlicht den Zusammenhang.
In dem Raum befindet sich eine konstante Anzahl von Gästen, die sich im Laufe der Zeit bewegen, um maximales Behagen zu erhalten.
Jede sich im Raum befindliche Person hat genau 8 Nachbarfelder. Sie bewegt sich auf dasjenige Feld, welches das kleinste Mißbehagen zur Folge hat. Sie verharrt an der momentanen Stelle, falls keines der Nachbarfelder ein gesteigertes Behagen verursacht.
(Miß–) Behagen wird durch geometrische Abstände beschrieben. Jeder Gast i hat zu jedem anderen Gast j einen bevorzugten Wunschabstand[1]. Der Mißbehagensfaktor (MBF) eines Gastes i wird durch folgende Summe beschrieben:

$$\text{MBF}(i) = \sum_j |\text{rD}(i,j) - \text{gD}(i,j)|$$

Anzumerken sei, daß sich die reale Distanz (rD) durch Anwendung des Satzes von Pythagoras ergibt (Euklidscher Abstand). Bei der gewünschten Distanz (gD) handelt es sich i.allg. um kein Abstandsmaß im mathematischen Sinne, d.h.

$$\text{gD}(i,j) \neq \text{gD}(j,i)$$

Der Partyablauf gliedert sich nun wie folgt: für jeden Gast (quasi–parallel) wird das Nachbarfeld (oder das momentane Feld) berechnet, welches den geringsten MBF aufweist. Anschließend bewegen (oder verharren) sich die Gäste auf dieses[2]. Dieser Iterationsschritt wird fortlaufend wiederholt. Interessant ist, ob eine Konvergenz des Systems zu beobachten ist, oder ob die Gäste bis in alle Zeit durch den Raum irren.

Die Daten für die Party sollen aus einer Datei beim Programmstart eingelesen werden. Das Format soll wie folgt sein:

```
# Groesse des Partyraumes:
```

[1] Besteht zu einem Gast kein gewünschter Abstand, zum Beispiel weil dieser einem nicht bekannt ist, so wird dieser in der nachfolgenden Berechnung **nicht** berücksichtigt.

[2] Quasi–Parallelität bedeutet in diesem Zusammenhand, daß **erst** für alle Gäste **das neue Feld berechnet** wird und **anschließend** diese **bewegt** werden.

```
+---------------------------------------+
|                                       |
|                                 A     |
|                                       |
|               F                       |
|              TTTTTTTT                 |
|              TTTTTTTT                 |
|                                  E    |
|                                       |
|                                       |
|       B                  C      D     |
+---------------------------------------+
```

Abbildung 11.1: Überblick über die Party

```
# Zeilen Spalten
15 30
# Start: Name Zeile Spalte
Albert 3 4
Berta  6 8
Caecilie 10 18
Dora 1 1
Emil 2 1
Frederike 7 25
# gewuenschte Distanz:
# Name Name Distanz
Albert Emil 10
Emil Albert 7
Dora Emil 23
Dora Berta 2
# Mobiliar: Zeichen Zeile Spalte
T 6 14
T 6 15
T 6 16
T 7 14
T 7 15
T 7 16
```

Zwei Personen dürfen sich nicht gleichzeitig auf einem Feld befinden. Das Tanzen auf den Tischen ist ebenfalls nicht erlaubt. Der Raum soll nicht verlassen werden.

Aufgabe 11.1

a Analysieren, entwerfen und implementieren Sie das beschriebene dynamische System.
Insbesondere beachten Sie folgende Aspekte:

- Entwerfen Sie die verschiedenen Objektklassen.
- Verwenden Sie geeignete (dynamische) Datenstrukturen, so daß das gestellte Problem einfach erweitert werden kann.
- Entwerfen Sie eine geeignete Bildschirmklasse `screen`.

Lektion 12

Vergleich von C++ und C

Wie bereits mehrfach erwähnt, ist C++ eine Erweiterung der Programmiersprache C (vgl. Abschnitt 1.1). Allerdings sind nicht alle Möglichkeiten und Eigenarten von C übernommen worden, d.h. nicht jedes C Programm ist automatisch auch ein gültiges und korrektes C++ Programm. Erschwert wird dieser Umstand weiterhin durch die verschiedenen C Dialekte, insbesondere **K&R C** sowie **ANSI–C**.

In den nachfolgenden Abschnitten wird auf die wichtigsten Unterschiede zwischen C++ und ANSI–C eingegangen. Soweit Differenzen zwischen dem klassischen K&R C existieren, wird auch auf diese kurz eingegangen.

12.1 Schlüsselwörter

ISO 2.8 In der nachfolgenden Tabelle 12.1 sind die Schlüsselwörter der Sprachen C und C++ aufgeführt[1]. Obwohl fast doppelt soviele **`Schlüsselwörter`** in C++ zur Verfügung stehen, werden diese nur einigen wenigen Situation benötigt.
In der ursprünglichen C Version von Kernighan und Ritchie sind die Schlüsselwörter `const`, `signed` und `volatile` **nicht** vorgesehen.

[1] In dieser Fibel sind nicht alle vorgestellt worden.

12.2 Abweichungen und Inkompatibilitäten

In den nachfolgenden Abschnitten wird konkret auf einige Unterschiede zwischen den Sprachen C und C++ eingegangen. Es sollen an dieser Stelle nur die wichtigsten aufgeführt werden, ansonsten sei auf die Literatur (z.B. [ISO95, Anhang C–2]) verwiesen.

Kommentar In C werden Kommentare mit `/*` und `*/` geklammert. Der Kommentar kann sich über mehrere Zeilen erstrecken. In C++ wird ein einzeiliger Kommentar mit einem doppelten Schrägstrich `//` eingeleitet. ISO C.2.1

```
int i;
  /* Definition der Variablen i
       (Kommentar in C und C++)
  */

int j;
  // Definition der Variablen j
  // nur in C++ erlaubt !
```

Gültigkeitsbereich `struct` In C++ definiert `struct` einen Gültigkeitsbereich (vgl. Abschnitt), d.h. die Bezeichner von Variablen, Typen etc können nur innerhalb der Struktur benutzt werden. Ansonsten muß mit dem binären Operator `::` der Gültigkeitsbereich explizit angegeben werden ISO C.2.2

asm	auto	**bool**	break
case	**catch**	char	**class**
const	**const_cast**	continue	default
delete	do	double	**dynamic_cast**
else	enum	**explicit**	extern
false	float	for	**friend**
goto	if	**inline**	int
long	**mutable**	**namespace**	**new**
operator	**private**	**protected**	**public**
register	**reinterpret_cast**	return	short
signed	sizeof	static	**static_cast**
struct	switch	**template**	**this**
throw	**true**	**try**	typedef
typeid	**typename**	union	unsigned
using	**virtual**	void	volatile
wchar_t	while		

Abbildung 12.1: Schlüsselwörter in C und C++

```
struct out {
 int value;
 struct in {
  double value; } str_in;
};

int main() {
 struct out X; /* C und C++ */
 struct in  Y; /* nur in C ! */

 struct out::in  Z; // in C++
}
```

ISO C.2.5 **Implizite Deklaration** In C werden alle **nicht deklarierten** Variablen und Funktionsrückgabewerte und –argumente, **implizit** als vom Typ `int` deklariert[2].

```
/* Die folgenden Konstrukte
   sind in C aber nicht in
   C++ erlaubt. */

void fun(const a);

main() {
 /* ... */
}

const i=1;
```

In C++ **muß** jede Deklaration mit einem Typ versehen werden. Die obigen Anweisungen lauten äquivalent daher

```
void fun(const int a);

int main() {
 /* ... */
}

const int i=1;
```

Beliebige Anzahl von Funktionsparametern
In C++ bedeutet in der Deklaration einer Funktion eine leere Liste von Parametern, daß die ISO C.2.6

[2] In [Koe95] bemerkt Andrew Koenig, daß ab der nächsten C Version, die zur Zeit erarbeitet wird, die implizite `int` Deklaration nicht mehr erlaubt sein wird. Der Unterschied dürfte also mit der nächsten Compilergeneration verschwinden.

Funktion **keine** Funktionsargumente hat. Hingegen bedeutet dieselbe Deklaration in C, daß die Anzahl und Typ der Parameter der Funktion nicht bekannt ist. Funktionen ohne Parameter müssen in C explizit mit `void` deklariert werden.

```
int fun();
  // C++ : Funktion ohne
  //       Argumente

int fun(void);
  /* C : Funktion ohne
         Argumente    */
```

ISO C.3.1 **Funktionsdeklaration im „alten Stil“** In C sind zwei Möglichkeiten zur Funktionsdefinition vorgesehen. Das nachfolgende Beispiel zeigt die unterschiedlichen Definitionsstile.

```
/* alter Stil */

int add(a,b)
 int a,b;
{
 return a+b;
}

/* neuer Stil */

int add2(int a,int b) {
 return a+b;
}
```

Im „alten Stil“ werden in der Parameterliste nur die Namen angegeben. Der Typ der einzelnen Parameter wird dann anschließend vor dem Funktionsrumpf deklariert. Der „neue Stil“ deklariert den Typ und Namen der Parameter zusammen in der Parameterliste der Funktion. In C sind beide Möglichkeiten erlaubt; trotzdem sollte der „neue Stil“, der ebenfalls ausschließlich in C++ Programmen erlaubt ist, bevorzugt werden.

Anhang A

C++ und C im Internet

In den folgenden Abschnitten werden einige Adressen im Internet aufgeführt, an denen C++ und C relevante Informationen zu finden sind.

Das Internet ist äußerst dynamisch und es ist schwer, wenn nicht sogar unmöglich, einen vollständigen und konsistenten Überblick über das Thema Programmieren in C und C++ zu geben. Es wird daher garnicht auch erst der Versuch gemacht.

Alle Adressen sind im **URL** (Uniform Resource Locator) Format angegeben.

A.1 Newsgroups

Die folgenden Newsgroups befassen sich mit Programmieren insbesondere mit C und C++ und einigen Werkzeugen.

`news:comp.lang.c++`
`news:de.comp.lang.c++` (deutsch)
Die objektorientierte Programmiersprache C++.

`news:comp.lang.c++.leda`
Aspekte über die LEDA Bibliothek. LEDA ist eine Klassenbibliothek, die die gebräuchlichsten Datentypen (z.B. Hashtabellen, Bäume, Graphen) enthält.

`news:comp.std.c++`
Diskussionen über die C++ Sprache, Bibliotheken und Standards.

`news:comp.lang.c.moderated`
Die C Programmiersprache (moderiert).

`news:comp.lang.c`
`news:de.comp.lang.c` (deutsch)
Diskussionen über C.

`news:comp.std.c`
Diskussionen über die Standardisierung von C.

Zu den GNUWerkzeugen existieren eine Reihe von Newsgroups mit dem Präfix `gnu`, z.B.

`news:gnu.emacs.help`
Fragen und Antworten zur Benutzung des GNU Emacs.

`news:gnu.g++.lib.bug`
Fehler in der GNU g++ Bibliothek.

Ein großer Vorteil der newsgroups ist die Möglichkeit der Kommunikation mit anderen und das weltweit. Bei Fragen und Problemen reicht es oftmals, eine kurze Anfrage in der entsprechenden newsgroup zu stellen. Eine Beantwortung erhält man i.d.R. schon nach ein oder zwei Tagen. Die newsgroups bieten die Möglichkeit an der aktuellen Diskussion (z.B. Standardisierung von C++) teilzunehmen.

A.2 World Wide Web

Das verteilte Informationssystem **World Wide Web** ist eine weitere Quelle von Informationen zum Thema Programmieren mit C und C++.

ANSI C Programming
School of Mathematical Sciences
University of Bath
P.J. Willis

`http://www.bath.ac.uk/~maspjw/NOTES/ansi_c/ansi_c.html`

Dieses Online Handbuch faßt die Programmiersprache C in der ANSI Version in Form eines Referenzhandbuchs zusammen.

C Programming
University of Strathclyde Computer Centre
Steve Holmes

`http://www.strath.ac.uk/CC/Courses/NewCcourse/ccourse.html`

Dieser Online C Programmierkurs wendet sich an Interessierte, die bereits Programmiererfahrungen in einer anderen Sprache haben.

C Programming
Central Institute of Technology, New Zealand
Brian Brown

`http://www.cit.ac.nz/smac/cprogram/default.htm`

Online Einführungkurs in die Programmiersprache C. Interessanterweise wird auch auf Hardware Aspekte eingegangen. Insbesondere ist eine Beschreibung der BIOS Systemaufrufe der 8088/86 CPU zu finden.

ANSI C for Programmers on UNIX Systems
Cambrigde University Engineering Department
Tim Love

`http://hpux.csc.liv.ac.uk/users/workexp/wk/c/cstuff/teaching_C/teaching_C.html`

Online Einführungskurs in ANSI C mit Schwerpunkt auf UNIX™ Systemen.

Frequently Asked Questions in comp.lang.c
Steve Summit

`http://www.lysator.liu.se/c/c-faq/index.html`

Das FAQ Dokument im html Format schildert die am häufigsten gestellten Fragen und Antworten.

The Free On-line Dictionary of Computing
Dennis Howe

`http://wombat.doc.ic.ac.uk/foldoc/index.html`

Unter dieser Adresse befindet sich ein suchbares Wörterbuch aus dem Bereich der Computerwissenschaften.

ISO Standardisation of C++
Volant Turnpike

`http://metro.turnpike.net/S/scorf/isocplus.html`

Diese Seite bietet einen Überblick über den Standardisierungsprozeß vo C++. Fernerhin werden Verweise (*links*) auf weitere C++ relevante Seiten im WWW gegeben.

Introduction to Object Oriented Programming Using C++
Macvicar School of Education and Technology
Globewide Network Academy
Marcus Speh

`http://uu-gna.mit.edu:8001/uu-gna/text/cc/index.html`

Einführender Online Kurs in C++ ausgezeichdet mit dem **Best of the Web '94 Award — Best Educational Service**.

The C++ Virtual Library
DESY
Marcus Speh, Lutz Lilje

`http://info.desy.de/user/projects/C++.html`

Hier befindet sich eine weitere Einstiegsseite zum Thema C++ und objektorientiertes Programmieren im Web.

User's Guide to the GNU C++ Library
Doug Lea

`http://csugrad.cs.vt.edu/manuals/libg/libg_toc.html`

Dieses Online Buch enthält eine etwas veralterte Beschreibung (1992) der GNU C++ Bibliothek (libg++).

Diverses
Cygnus Support
Mike Stump

`http://www.cygnus.com/~mrs/`

Diverses (z.B. Liste aller bekannten Fehler des `g++`) und aktuelle Informationen zum `gcc` und zur Standard Template Library (STL).

C++ Annotations
State University of Groningen
Frank B. Brokken, Karel Kubat

`http://www.icce.rug.nl/docs/cplusplus/cplusplus.html`

Dieses Online Handbuch ist für Programmierer und Programmiererinnen gedacht, die bereits erste Erfahrungen mit der Sprache C++ gemacht haben. Es werden die u.a. die Begriffe Polymorphismus und Schablone (Template) anhand von konkreten Programmbeispielen erklärt.

The Standard Template Library
Rensselaer Polytechnic Institute

`http://www.cs.rpi.edu/~musser/stl.html`

Die jüngsten Informationen inklusive der freiverfügbaren STL Implementierung von Hewlett–Packard sind hier zu finden. Es wird einführend die Benutzung der Standard Template Library beschrieben.

A.3 Anonymous FTP–Server

Nachfolgend einige FTP–Server,[1] auf denen weitere Informationen oder Software zum Themengebiet C und C++ zu finden sind.

Um den Datenverkehr und die Kosten gering zu halten, sollten die nächstmöglichen gespiegelten FTP–Server genutzt werden.

GNU

`ftp://prep.ai.mit.edu/pub/gnu`

Der offizielle GNU Server, auf dem sich die hier beschriebenen Werkzeuge (z.B. `gcc` und `emacs`) befinden.

ISO Draft Standard C++

`ftp://research.att.com/dist/stdc++/WP`

Hier befindet sich der Entwurf für den C++ Standard im Postscript oder PDF Format.

Learn C and C++ Today
Vinit Carpenter

`ftp://rtfm.mit.edu/pub/usenet/news.answers/C-faq/learn-c-cpp-today`

Dieses Dokument enthält viele Hinweise, wo im Internet Informationen zu C und C++ zu finden sind. Es werden auch einige C++ Bücher besprochen.

[1] Auf diesen Rechnern wird mit `anonymous` als Name und der eigenen email Adresse als Paßwort sich angemeldet.

Anhang B

Musterlösungen

Nachfolgend werden zu ausgewählten Aufgaben mögliche Lösungen angegeben.

B.1 Lektion 2 — Umgebung

Aufgabe 2.3
a Die Kommandos zum Ziel `z1` werden in zwei verschiedenen Shells sequentiell abgearbeitet. Die erste Shell wechselt in das Verzeichnis `/tmp` und beendet sich dann. Die zweite Shell gibt dann den lokalen Pfad aus, in dem `make` aufgerufen wurde.
`z2` wird in einer einzigen Shell abgearbeitet. Dort wird zunächst in das Verzeichnis `/tmp` gewechselt und dann dieser Pfadname ausgegeben.

b Ist die Abhängigkeit von Ziel `x.c` erfüllt, d.h. `x.h` ist jünger als `x.c`, wird durch den `touch` Befehl die Änderungszeit von der Datei `x.c` auf den aktuellen Zeitpunkt verändert.
Dieses hat zur Folge, daß `x.c` neu übersetzt werden muß, da folglich das Ziel `x.o` veraltet ist. Entsprechendes gilt für `y.h`, `y.c` und `y.o`. Durch diesen Mechanismus ist es möglich zu veranlassen, daß eine `.c` Datei neu kompiliert wird, wenn die zugehörige `.h` Datei verändert worden ist. Dieses läßt sich natürlich auf eine Abhängigkeit einer Liste von `.h` Dateien erweitern.

Aufgabe 2.4

a Bedeutung des `#` Operators: ISO 16.3.2
Der entsprechende Parameter in einem funktionalen Makro wird als Zeichenkette interpretiert, d.h., bevor eine Ersetzung vorgenommen wird, wird diese mit `"` geklammert.
Beispiel:

```
#define debug(s)   \
        printf( #s "\n")

int main() {
 debug(error);
}
```

wird zu

```
int main() {
 printf("error\n");
}
```

ersetzt.
Bedeutung des `##` Operators: ISO 16.3.3
Dieser Operator dient in der Makrodefiniton zur Verkettung von Symbolen.
Beispiel:

```
#define TEMP(i)     temp ## i
TEMP(1) = TEMP(2);
```

Nach dem Durchlauf des Präprozessors ergibt sich dann

```
temp1 = temp2;
```

b Die erste C/C++ Funktion ermittelt das Quadrat einer ganzen Zahl. Dabei wird automatisch beim Übersetzungsvorgang eine Typkontrolle durchgeführt. Das Makro bewirkt eine rein syntaktische Ersetzung im Präprozessordurchlauf. Hier liegt auch eine mögliche Fehlerquelle.

```
int a=square(5+3);
 // berechnet 64
int b=SQUARE(5+3);
 // Fehler wegen
 // 5+3*5+3 = 23
```

Hieraus ergibt sich, daß in der Makrodefinition **grundsätzlich** alle Parameter geklammert werden sollten.

c Falls `_NAME_OF_HEADERFILE` als Name für ein Makro noch nicht existiert, wird dieses definiert und die nachfolgenden Anweisungen durchgeführt. Bei einer Einbindung von Quelltexten, die nicht diesem Schema entsprechen, kann es zu *doppelten* Deklarationen und Definitionen kommen, die beim Übersetzungsvorgang bemängelt werden.
Headerdatei `prog1.h`

```
int x;
```

Headerdatei `prog2.h`

```
#include "prog1.h"

int y;
x=47;
```

Das Hauptprogramm liest die beiden Headerdateien ein.

```
#include <iostream.h>
#include "prog1.h"
#include "prog2.h"

int main() {
 y=11;
 cout << x << y << endl;
}
```

Der Binder meldet, daß die Variable `x` mehrfach definiert wurde. Abhilfe ist durch die Verwendung einiger Makros möglich. Headerdatei `prog1.h` mit Makro

```
#ifndef _PROG1_H
#define _PROG1_H
int x;
#endif
```

Headerdatei `prog2.h` mit Makro

```
#ifndef _PROG2_H
#define _PROG2_H

#include "prog1.h"

int y;
x=47;
#endif
```

d Bei `__LINE__` und `__FILE__` handelt es sich um besondere Makros. Diese Makros können nicht umdefiniert oder ausgeschaltet werden. Weiterhin ist es nicht nötig, zu ihrer Verwendung weitere Definitionsdateien einzubinden. ISO 16.8
`__LINE__` gibt die Zeilennummer des Quelltextes an, in der dieses Makro steht. `__FILE__` beinhaltet den Namen der Quelldatei, in der dieses Makro steht. `__DATE__` bzw. `__TIME__` definieren die Zeitpunkte des Starts der Übersetzung.

B.2 Lektion3 — Fehlersuche und Testhilfen

Aufgabe 3.1

a Korrekt lautet das Programm:

```
int main() {
  char *a, *b;
  a=new char[30];

  b=a;
}
```

Der Dereferenzierungsoperator `*` bezieht sich nur auf die jeweils nachfolgende Variable. Somit ist die Variable `b` im ursprünglichen fehlerhaften Programm eine Variable vom Typ `char` und nicht ein `char*` Zeiger.

b Der Schalter `-D` ist eine Präprozessoranweisung im Makefile und entspricht einer `#define` Anweisung im Quelltext. Diese Anweisung erzeugt und belegt eine Variable für den Präprozessor mit dem angegebenen Wert. Falls kein Wert angegeben wird, wird die Variable standardmäßig mit 1 belegt.

`TRUE` wird definiert und auf `1` gesetzt.

Aufgabe 3.3

a Der Befehl für bedingte Unterbrechungspunkte im `xxgdb` Debugger lautet:

`condition` *N COND*, wobei *N* eine Zahl und *COND* die Bedingung ist. *N* gibt dabei die Nummer der zuvor gesetzten Unterbrechnung an.

b Variablen können durch den Befehl `set` *VAR=EXP* oder durch `set variable` *VAR=EXP* gesetzt werden.

c Benutzereigene Befehle lassen sich über den Befehl `define` definieren. Wobei direkt nach `define` der Befehlsname folgt. Anschließend kann dann pro Zeile ein Befehl eingegeben werden. Beendet wird die Definition durch ein `end` in einer Zeile.

Aufgabe 3.5

a und **b** Quelltext

```
#include<stdio.h>
#include<iostream.h>
#include<fstream.h>

class trace {
   ostream* to;

  public :

   trace(ostream& out =cout,
    char* text ="") : to(&out) {
     (*to) << text << endl;
   }

   ~trace() {
     (*to)<<"Ende"<<endl;}
};
```

```
main(int argc,char* argv[]) {
 if(argc==3) {
  ostream *out=
       new ofstream(argv[1]);
  trace x(*out,argv[2]);
  delete out;
 }
 else
  if(argc==2)
   trace y(cout, argv[1]);
}
```

c Diese Anweisung erzeugt dynamisch eine Variable. Da kein Destruktor am Ende der Funktion aufgerufen wird, erfolgt keine Freigabe des beanspruchten Speicherplatzes. Infolgedessen wird auch keine Kontrollausgabe ausgegeben; der eigentliche Sinn, nämlich die Kontrolle des Programmablaufs, wäre somit verloren.
Der Destruktor würde aufgerufen, wenn mit `delete` der benutzte Speicherbereich explizit freigegeben wird.

Aufgabe 3.6

a Durch den geschachtelten ternären Operator `:?` wird zuerst überprüft, ob die Bedingung `condition` erfüllt ist. Falls nicht, wird dann anschließend überprüft, ob das Gewicht `severity` größer als der Schwellwert ist. Falls dieses der Fall ist, wird eine Fehlermeldung ausgegeben. Mit Hilfe des # Operators ist es möglich, die Bedingung als Zeichenkette zu interpretieren.
Falls die Bedingung `condition` erfüllt ist, wird `1` ansonsten `0` (Kommaoperator !) zurückgegeben.

Aufgabe 3.7

a Beispiel

```
class something {
  char* mem;
 public :

  /* irgendetwas */
```

```
};
```

Ein korrekt arbeitender Zuweisungsoperator muß nicht `mem` kopieren, sondern den allokierten Speicher, auf den `mem` zeigt. Zuvor sollte aber der Speicherbereich, allokiert vom Objekt links des Zuweisungszeichens, wieder freigeben werden. Bei einer Selbstzuweisung der Form `s=s` muß jedoch verhindert werden, daß der Speicherbereich freigegeben und dann anschließend kopiert wird, da dieses u.U. zu einem Speicherfehler zur Laufzeit führen kann.

```
something& something::operator=(
 const something& s) {
  /* Selbstzuweisung ? */
  if (this!=&s) {
   /* Freigeben  */
   delete mem;
   /* Allokieren */
   mem=new char;
   /* Kopieren   */
   *mem=s->mem;
  }
 return *this;
}
```

B.3 Lektion 5 — Einführung

Aufgabe 5.1

ISO 2.7 **a** C++ Regeln für Namensgebung:

- Die Namen dürfen ausschließlich aus alphanumerischen Zeichen und Unterstrichzeichen _ bestehen.
- Das erste Zeichen eines Namens darf keine Ziffer sein.
- Es wird zwischen Groß- und Kleinschreibung unterschieden.
- Die Verwendung eines C++ Schlüsselwortes als Name ist nicht erlaubt.
- Es gibt keine Beschränkung der Namenslänge; alle Zeichen sind signifikant.

b Standardtypen (vgl. auch [HS91, Seite 122]) in C++

- `void`
- Skalare Typen
  - Zeigertypen
  - Arithmetrische Typen
    - Gleitkommatypen
    - Ganzzahltypen
      - Aufzählungstypen
      - Zeichentypen
  - Boolescher Typ (**nur in C++ verfügbar**)
- Funktionstypen
- Uniontypen
- Zusammengesetzte Typen
  - Vektortypen
  - Strukturtypen

c Folgende Tabelle faßt die Operatoren und Präzedenzen in C++ (vgl. [Str91, Seite 89] und [Lip91, Seite 84]) zusammen.

Symbol	Operator	
`::`	Gültigkeitsbereichsauflösung (Klasse)	r
`::`	Gültigkeitsbereichauflösung (global)	l
`.`	Komponentenauswahl	l
`->`	Komponentenauswahl	l
`[]`	Feld Indexierung	l
`()`	Funktionsaufruf	l
`()`	Typkonstruktion	l
`++`	Inkrementierung (post)	r
`--`	Dekrementierung (post)	r
`sizeof`	Größe	r
`++`	Inkrementierung (prä)	r
`--`	Dekrementierung (prä)	r
`~`	Komplement	r
`!`	Nicht	r
`-`	Minus (unär)	r
`+`	Plus (unär)	r
`&`	Adresse	r
`*`	Dereferenzierung	r
`new`	Allokation	r
`delete`	Freigabe	r
`.*`	Zeigerkomponentenauswahl	l
`->*`	Zeigerkomponentenauswahl	l
`*`	Multiplikation	l
`/`	Division	l
`%`	Modulo	l
`+`	Addition	l
`-`	Subtraktion	l
`<<`	Links Schieben	l
`>>`	Rechts Schieben	l
`<`	Kleiner	l
`<=`	Kleiner gleich	l
`>`	Größer	l
`>=`	Größer gleich	l
`==`	Gleich	l
`!=`	Ungleich	l
`&`	Bitweises Und	l
`^`	Bitweises Exklusiv Oder	l
`|`	Bitweises Oder	l
`&&`	Logisches Und	l
`||`	Logisches Oder	l
`?:`	Bedingte Auswertung	r
`=`	Zuweisung	r
`*=`	Multiplikation Zuweisung	r
`/=`	Division Zuweisung	r
`%=`	Modulo Zuweisung	r
`+=`	Addition Zuweisung	r
`-=`	Subtraktion Zuweisung	r
`<<=`	Links Schieben Zuweisung	r
`>>=`	Rechts Schieben Zuweisung	r
`&=`	Bitweises Und Zuweisung	r
`|=`	Bitweises Oder Zuweisung	r
`^=`	Bitweises Ex Oder Zuweisung	r
`,`	Sequenz	l

Der Eintrag in der dritten Spalte gibt die Assoziativität wieder (r=rechts,l=links).

d `?:` ist ein Bedingungsoperator. Entspricht aber nicht der `if`–Anweisung. Es ist der einzige ternäre Operator.

e Außer `sizeof` gibt es keine weiteren.

f `'a'` ist vom Typ `char`, während `"a"` ein `char[2]` Feld mit den beiden Elementen `'a'` und `'\0'` darstellt.

g Die Speicherklasse eines Objektes hat Einfluß auf die folgenden Eigenschaften:

Bindung bestimmt, welcher Bezeichner zu welchem Objekt gehört.

Lebensdauer bestimmt die Dauer, in der ein Objekt im (Haupt–) Speicher gehalten wird. Dieses bestimmt ebenfalls den Aufruf von Konstruktoren und Destruktoren von Objekten.

Gültigkeitsbereich gibt an, in welchem Quelltextbereich ein Objekt unter dem gebundenen Bezeichner anzusprechen ist.

ISO 7.1.1 Eines der vier folgenden Schlüsselwörter legt bei der Deklaration die Speicherklasse eines Objektes fest.

`extern` Objekten, die mit `extern` deklariert werden, wird kein Speicher zugewiesen, d.h. im Gegensatz zu den drei folgenden Spezifizierern handelt es also um keine **Definition**. Diese hat an anderer Stelle im Programm zu erfolgen.

`static` Ein als `static` deklariertes Objekt hat eine statische, d.h. oftmals über die Laufzeit des Programms andauernde Speicherung. Das folgende Beispiel verdeutlicht dieses:

```
int count(bool i =false) {
 static int x;
 x=(i? 0 : x+1);
 return x;
}

int main() {
 count(true)
 for(register int i=0;
     i<10;i++)
  cout << count() << endl;
}
```

Die Variable `x` wird während des gesamten Programmablaufs im Speicher gehalten, auch wenn der umgebene Block (hier die Funktion `count`) verlassen wird. Der Gültigkeitsbereich ist jedoch lokal in der Funktion. Das aufgeführte Programm gibt die Zahlen `1` bis `10` auf den Bildschirm aus.

`auto` Objekte dieser Speicherklasse haben eine automatische Speicherdauer, d.h. es wird solange im Speicher gehalten, bis der umgebene Block verlassen wird. Das Schlüsselwort `auto` ist redundant; wird keines der drei anderen explizit benutzt, ist das deklarierte Objekt implizit als `auto` deklariert.

`register` Ist ein Objekt dieser Speicherklasse zugewiesen, so wird der Übersetzer angewiesen, das Objekt in ein Register der CPU und nicht an einer Adresse im Hauptspeicher abzulegen. Dieses kann erhebliche Geschwindigkeitsvorteile haben. Allerdings ist der Kompiler **nicht** gehalten, diese Aufforderung zu befolgen. Oftmals ist es nämlich aus verschiedenen Gründen nicht möglich, Variablen in der CPU zu speichern. Ansonsten gelten die gleichen Regeln der `auto` Speicherklasse.

h Es gibt vier verschiedene Gültigkeitsbereiche: ISO 3.3.1

Lokal Ein Name, der in einem durch `{` und `}` geklammerten Block deklariert ist, kann nur innerhalb dieses Blocks benutzt werden. ISO 3.3.3

Funktion Ein **label**[1] (Sprungadresse) ist innerhalb der Funktion gültig, in der dieses definiert wurde. ISO 3.3.5

Klasse Der Name einer Variable oder Funktion in einer Klasse ist nur dort oder in abgeleiteten Klassen gültig. Fernerhin ist es möglich, auf einzelne Bezeichner mit dem `.`oder `->` Operator zuzugreifen. Es sind hierbei allerdings die Zugriffsqualifizierer `private`, `protected` und `public` zu beachten.

Datei Alle Namen, die nicht in eine der bisherigen drei Kategorien fallen, sind **global**, d.h. innerhalb der entsprechenden Datei, gültig. Da oftmals durch `#include` Anweisungen weitere Dateien beim Übersetzungsvorgang eingebunden werden, sind globale Namen auch in den nachfolgend eingebundenen Dateien gültig.

[1] labels sind in Verbindung mit der `goto` Anweisung von Bedeutung. `goto` Anweisungen sind i.allg. überflüssig und erschweren die Lesbarkeit von Quelltexten. Daher wird hier auch nicht weiter auf solche Konstrukte eingegangen.

ISO 7.2 **i** Aufzählungstypen können mit enum realisiert werden, z.B.

```
enum Farbe {rot, blau, gelb};
```

Die einzelnen Elemente werden als Konstanten behandelt. Wird nichts angegeben, so werden die Elemente, bei null beginnend, fortwährend von links nach rechts aufsteigend durchnumeriert. Mit dem Zuweisungsoperator = können jedoch auch andere Werte vereinbart werden.

```
enum Farbe {rot =7, blau =6,
            gelb =5};
```

Es ist möglich, einen Aufzählungstyp in einen `int` Typ umzuwandeln und umgekehrt. Im zweiten Fall hat dieses explizit zu erfolgen.

```
int i=rot;    // implizit
Farbe j=gelb;
j=Farbe(i);  // explizit
```

ISO 3.9.2 **j** `void*` ist ein Zeiger auf eine Struktur eines unbestimmten Typs. Ein Zeiger auf ein nicht konstantes Objekt kann in einen `void*` Zeiger umgewandelt werden.

ISO 8.3.4 **k** In C und C++ lassen sich Felder beliebiger endlicher Dimension definieren.

```
int vektor[7]={0,1,2,3,4,5,6};
```

ISO 8.5.1 definiert ein eindimensionales Feld mit 7 Elementen. Zusätzlich wird das Feld mit den `int` Werten 0 bis 6 initialisiert, d.h. `vektor[0]` wird `0`, `vektor[1]` wird `1`, usw. zugewiesen. Ist die Initialisiererliste kleiner als das Feld, so werden die restlichen Elemente mit Nullen des entsprechenden Typs aufgefüllt.

```
int matrix[3][2]={{1,2},{3,4},
                  {5,6}};
```

definiert ein zweidimensionales Feld mit $3{\times}2$ Elementen, das sind drei Zeilen und zwei Spalten. Entsprechend des `vektor` Beispiels werden die Elemente des Feldes initialisiert.
Die Klammerung der einzelnen Zeilen ist redundant, d.h. mit den folgenden Anweisungen wird der gleiche Effekt erzielt.

```
int matrix[3][2]={1,2,3,4,5,6};
```

Eine Besonderheit nehmen Zeichenketten (Strings) ein. Anstatt ein `char` Feld mit ISO 8.5.2

```
char carr[11]=
     {'I','n','f','o','r','m',
       'a','t','i','k','\0'};
```

zu initialisieren, ist die folgende vereinfachte Schreibweise möglich.

```
char carr[]="Informatik";
```

Zu beachten ist, daß im zweiten Fall automatisch ein `'\0'` als Zeichenkettenabschluß angefügt wird, d.h. das Feld um eins größer ist als der mit `"` geklammerte String.

l

```
typedef const int cInt;
 // Konstante Ganzzahlen

typedef float* fPtr;
 // Zeiger auf kurze
 // Realzahlen

typedef const char* cCharPtr;
 // Zeiger auf konstante
 // Zeichen

typedef char* const CharPtrc;
 // konstante Zeiger auf
 // Zeichen

typedef const char* const
        cCharPtrc;
 // konstante Zeiger auf
 // konstante Zeichen

typedef unsigned long
        uLong6Array[6];
 // Feld mit 6 positiven
 // langen Ganzzahlen

int main() {
 float a=69.0;
 const char b='M';
 char c='U',d='K';
```

```
    cInt x=42;
    fPtr y=&a;
    cCharPtr r=&b;
    uLong6Array w={1,2,3,4,5,6};
    CharPtrc z=&c;
    cCharPtrc v=&d;

    *r=c;  // Fehler !
     z=&d; // Fehler !
  }
```

ISO 18.4.1.1 **m** Der Operator `new` allokiert Speicher für ein Objekt und gibt einen Zeiger des entsprechenden Typs zurück.

```
int* f() {
 int y=7,*x=new int;
 *x=y;
 return x;
}

main() {
 int* z=f();
 delete z;
}
```

Die Lebensdauer eines mit `new` erzeugten Objekts ist nicht auf den Gültigkeitsbereich des momentanen Bezeichners beschränkt. Im aufgeführten Beispiel wird in der Funktion `f` lokal ein `int` Objekt `y` erzeugt und mit sieben initialisiert. `y` ist lokal in der Funktion gültig, d.h. nach Rücksprung aus der Funktion wird der allokierte Speicher gelöscht und kann von anderen Objekten erneut allokiert werden. Hingegen ist `x` ein Zeiger auf ein `int` Objekt, welches **dynamisch** erzeugt wird. Nach Rücksprung aus `f` ist der Name `x` nicht mehr gültig, das referenzierte Objekt jedoch ist weiterhin im Speicher und kann in nachfolgenden Anweisungen benutzt werden. Die Lebensdauer eines mit `new` allokierten Objektes endet mit der **expliziten** Freigabe durch einen `delete` Operator.

Aufgabe 5.2

ISO 6.6.1 **a** In Schleifen und innerhalb der `switch` Anweisung verläßt der `break` Befehl in umgebene Schleife bzw. die umgebene `switch` Anweisung. Eine fehlende `break` Anweisung ist oftmals eine Fehlerquelle.
Beispiel:

```
int main() {
 enum Farbe {rot, blau, gelb};
 Farbe option;
           // beliebiger Wert

 switch (option) {
  case rot  : printf("rot ");
  case blau : printf("blau ");
  case gelb : printf("gelb ");
  default   : printf("??? ");
 }
}
```

Abhängig des Wertes der Variable `option` soll auf dem Bildschirm ein Farbwert ausgegeben werden.
Hat in diesem Beispiel `option` den Wert `rot`, so wird `rot blau gelb ???` ausgegeben, was nicht beabsichtigt ist. Gilt `option==blau`, so wird `blau gelb ???` ausgegeben.
Bei der Abarbeitung der `switch` Anweisung wird zur korrekten Einsprungadresse (case label) gesprungen und dann die nachfolgenden Anweisungen ausgeführt. Weitere Einsprungadresse haben also keinen Effekt auf die Ausführung. Dieses kann man sich, wie folgendes Beispiel verdeutlicht, zu Nutze machen.

b

```
switch (a) {
 case 1  : tag="Montag";
           break;
 case 2  : tag="Dienstag";
           break;
 case 3  : tag="Mittwoch";
           break;
 case 4  : tag="Donnerstag";
           break;
 case 5  : tag="Freitag";
```

```
              break;
 case 6   : tag="Samstag";
            break;
 case 0   :
 case 7   : tag="Sonntag";
            break;
 default  : tag=(char*)0;
}
```

ISO 6.5.1 **c** In einer `while` Schleife werden solange die nachfolgenden Anweisungen ausgeführt, bis die Bedingung `false` ist. Der Test findet **vor** jeder Ausführung statt.

ISO 6.5.3 In einer `do while` Schleife werden ebenfalls die Anweisungen ausgeführt, bis die dem `while` folgende Bedingung falsch ist. Der Test befindet jedoch **nach** jeder Ausführung statt, d.h. die Schleife wird mindestens einmal durchlaufen. Dieses Verhalten entspricht **nicht** einer *repeat until* Schleife in Modula–2, da dort die Schleife terminiert, wenn die Bedingung `true`, also wahr, ist.

d Dieses Konstrukt entspricht einer nicht terminierenden Schleife (Endlosschleife).

Aufgabe 5.3

a Bei einem Funktionspointer handelt es sich einen Zeiger auf eine Funktion.
Beispiel:

```
void error(char* text) {
 cout << "error: " << text
      << endl;
}

main() {
 void (*pf)(char*)=&error;

 (*pf)("overflow");
}
```

`pf` zeigt auf eine Funktion mit einem `char*` als Argument. Da der Klammeroperator `()` eine höhere Präzedenz hat, muß der Derefenzierungsoperator `*` geklammert werden, um eine korrekte Übersetzung zu ermöglichen.

b Um einen unnötigen Funktionsaufruf zu sparen, ist es möglich, mit der `inline` Anweisung den Funktionsrumpf zu expandieren, d.h. der Aufruf und der Rücksprung der entsprechenden Funktion wird vermieden. ISO 7.1.2
Anstatt dessen fügt der Übersetzer bei jedem Aufruf die Anweisungen des Funktionsrumpfes der mit `inline` spezifizierten Funktion in den Quelltext ein.
Bei der Anweisung handelt es sich allerdings um einen Hinweis, der vom Kompiler ignoriert werden kann.

```
void init(int& x) {
 x=0;
}
```

Bei einem Funktionsaufruf werden intern die folgenden Aktionen durchgeführt:

Rücksprungadresse merken
Register sichern
Sprung an den Beginn der Funktion
`x=0;`
Register zurücksetzen
Sprung an die Rücksprungadresse

Bei einer Funktionsdeklaration mit `inline` fallen alle Verwaltungsschritte (*kursiv* dargestellt) weg. Hierdurch wird das vom Kompiler erzeugte Programm i.allg. länger, da für jeden Funktionsaufruf der entsprechende Rumpf expandiert wird. Das Programm kann jedoch erheblich schneller werden, da u.U. viele Verwaltungsschritte eingespart werden.

c

```
#include<stdio.h>

char* kopieren(int anfang,
 int ende, char* text) {
  char* ziel;
  ziel=new
       char[ende-anfang+2];
  for(int i=0;i<ende-anfang+1;
      i++)
```

```
   ziel[i]=text[i+anfang];
  ziel[ende-anfang+1]='\0';
  return ziel;
}

int main() {
 printf("%s\n",
  kopieren(3,6,"testtext"));
}
```

d

```
#include<stdio.h>

int main(int argc, char* argv[]) {
 printf("Moin, Moin");
 if (argc!=1)
  for (int i = 1; i<argc; i++)
   printf( %s", argv[i]);

 printf("\n");
}
```

e Nachfolgend wird anhand eines Beispiels gezeigt, wie die Ellipse `...` benutzt werden kann, um in einer Funktion eine beliebige Anzahl von Argumenten zuübergeben.

```
#include <stdarg.h>

double mean(int n,...) {
 double m=0.0;

 va_list args;
  // Argumentenliste
  // deklarieren

 va_start(args,n);
  // Initialisieren

 for(int i=0;i<n;i++)
  m+=(double)
     va_arg(args,double);
   // Argument liefern

 va_end(args);
  // Argumentliste freigeben

 return m/n;
}
```

```
int main() {

 double a=
  mean(3,10.0,20.0,30.0);
 double b=
  mean(5,1.1,2.2,3.3,4.4,5.5);

}
```

f Eine weitere Möglichkeit, Objekte ohne einen zeitaufwendigen Kopiervorgang in Funktionen zu übergeben, ist durch die Verwendung von Zeigern auf die Objekte gegeben.

Aufgabe 5.4

e

```
#include <iostream.h>

class boolean {
  int value;
  boolean isEqual(int) const;

  static const char* strFalse;
  static const char* strTrue;

 public:

  boolean();
  boolean(int);
  boolean(const boolean&);

  boolean& operator=(const boolean&);

  void set(const boolean&);
  void set(int);

  operator const char*() const;
  operator int() const;

  void print(ostream& =cout) const;

  boolean operator! () const;

  boolean operator&& (const
              boolean&) const;
```

```
  boolean operator|| (const
              boolean&) const;

  boolean operator== (const
              boolean&) const;

  boolean and(const boolean&)
              const;

  boolean or(const boolean&)
              const;

  boolean nand(const boolean&)
              const;

  boolean xor(const boolean&)
              const;

  friend boolean operator==(int,
              const boolean&);

  friend boolean operator==(const
              boolean&,int);

  friend boolean not(const
              boolean&);

};

const char*
 boolean::strFalse="false";
const char*
 boolean::strTrue="true";

inline void boolean::set(int val) {
 value=(val ? 1 : 0);
}

inline void boolean::set(const
 boolean& b) {
  value=b.value;
}

inline boolean& boolean::operator=(
 const boolean& b) {
  value=b.value;
  return *this;
}
```

```
inline boolean boolean::operator==(
 const boolean& b) const {
  return boolean(value==b.value);
}

inline boolean::operator int()
 const {
  return value;
}

inline boolean::operator const char*
 () const {
  return
   (value? strTrue : strFalse);
}

inline boolean::boolean():value(1)
{}

inline boolean::boolean(int i) {
 set(i);
}

inline boolean::boolean(const
 boolean& b) : value(b.value) {}

inline void boolean::print(ostream&
 out) const {
  out << (char*)(*this) << flush;
}

boolean boolean::isEqual(int i)
 const {
  if (value==i)
   return boolean(1);
  else
   if (value == 1 && i>1)
    return boolean(1);

 return boolean(0);
}

inline boolean boolean::and(const
 boolean& b) const {
  return boolean(value && b.value);
}

inline boolean boolean::or(const
 boolean& b) const {
```

```
  return boolean(value || b.value);
}

inline boolean boolean::xor(const
 boolean& b) const {
  return boolean(value && !b.value
           || !value && b.value);
}

inline boolean boolean::nand(const
 boolean& b) const {
  return not(and(b));
}

inline boolean boolean::operator&&(
 const boolean& b) const {
  return and(b);
}

inline boolean boolean::operator||(
 const boolean& b) const {
  return or(b);
}

inline boolean boolean::operator!()
 const {
  return not(*this);
}

inline boolean operator==(int i,
 const boolean& b) {
  return b.isEqual(i);
}

inline boolean operator==(const
 boolean& b, int i) {
  return b.isEqual(i);
}

inline boolean not(const boolean&
 b) {
  return boolean(!b.value);
}
```

g

```
class complex {
  double real,imag;

 public:

  complex(double r=.0,double i=.0);

  void print(ostream& =cout) const;

  complex operator+(const complex&)
          const;

  complex operator-(const complex&)
          const;

  complex operator*(const complex&)
          const;

  complex operator/(const complex&)
          const;

  friend
   complex conj(const complex&);

};

inline complex::complex(double r,
 double i) : real(r),imag(i) {}

inline void complex::print(ostream&
 out) const {
  out << real
      << (imag<0.0 ? "" : "+")
      << imag << 'i' << endl;
}

inline complex complex::operator+(
 const complex& c) const {
  return
   complex(real+c.real,imag+c.imag);
}

inline complex complex::operator-(
 const complex& c) const {
  return
   complex(real-c.real,imag-c.imag);
}

inline complex complex::operator*(
 const complex& c) const {
  return
   complex(real*c.real-imag*c.imag,
          real*c.imag+c.real*imag);
```

```
}

inline complex complex::operator/(
 const complex& c) const {
  double
   diff(c.real*c.real+c.imag*c.imag);
  return
   complex((real*c.real+imag*c.imag)
        /diff,
        (c.real*imag-real*c.imag)
        /diff);
}

inline complex conj(const complex&
 c) {
  return
   complex(c.real,-1.0*c.imag);
}
```

Aufgabe 5.5

ISO 27.4.3.1.2 **a** Die drei Manipulatoren `dec`, `oct` und `hex` dienen zur Konvertierung eines Objekts vom Typ `int` in eine dezimale, oktale oder hexadezimale Darstellung.

```
cout << dec << 1204 << '\n'
     << oct << 1204 << '\n'
     << hex << 1204 << '\n'
     << flush;
```

ISO 27.6.3 **b**

```
cout << setw(11)
     << setfill('*')
     << '!' << endl;
```

c `'\n'` fügt ein Neue–Zeile Zeichen in den Ausgabestrom ein. Hingegen hängt `endl` zusätzlich ein `flush` an, um den Ausgabestrom auch wirklich auszugeben.

d Die Löschung des Bildschirms ist durch einen Trick möglich. Der `system` Aufruf öffnet eine Shell und führt das als Parameter übergebene Kommando in dieser aus[2].

[2] `clear` ist ein UNIX™ Kommando zum Löschen des Bildschirms.

```
   system("clear");
```

Aufgabe 5.6

a Der Manipulator `ws` sorgt dafür, daß sogenannte **whitespaces**, also Tabulator–, Leer– und Neue–Zeile Zeichen bei der Eingabe ignoriert werden.

b

```
cin.operator>>(x).operator>>(y);
```

Aufgabe 5.7

a Die folgenden logischen Abfragefunktionen ermöglichen es, den Status des Eingabestroms abzufragen. ISO 27.4.4.3

`good()` liefert `true`, falls kein Fehler bei der Eingabe aufgetreten ist.

`eof()` liefert `true`, falls das Ende der Eingabesequenz erreicht wurde.

`fail()` liefert `true`, falls bei der Eingabe ein Fehler, z.B. durch ein fehlerhaftes Medium oder ein unerwartetes Zeichen beim Einlesen, aufgetreten ist.

`bad()` liefert `true`, falls bei der Eingabe ein Fehler durch ein fehlerhaftes Medium aufgetreten ist.

`operator bool()` liefert `true`, falls kein Fehler aufgetreten ist, d.h. falls `fail()==false`.

`operator !()` liefert `true`, falls ein Fehler aufgetreten ist, d.h. falls `fail()==true`.

b und **c** Es gibt keine Unterschiede. Das Ergebnis ist jeweils

```
[****12]
[12]
```

Aufgabe 5.8

a Mit den folgenden Einstellungen ist es möglich, eine Datei zum Lesen oder Schreiben zu öffnen. Die einzelnen Bits können durch ein **bitweises Oder** verknüpft werden.

`ios::app` fügt am Ende bei jedem Schreibzugriff an.

`ios::ate` positioniert ans Ende der Datei.

`ios::binary` binärer Lese– bzw. Schreibmodus.

`ios::io` öffnet zum Lesen.

`ios::out` öffnet zum Schreiben.

`ios::trunc` löscht eine bereits existierende Datei beim Öffnen der Datei zum Schreiben.

Aufgabe 5.9

ISO 3.6.1 **a** Jedes Programm muß eine Funktion

```
int main() { /* ... */ }
```

oder

```
int main(int argc,
         char* argv[]) {
 /* ... */
}
```

definieren. Im zweiten Fall gibt `argc` die Anzahl der Argumente `argv` an. Per Definition steht in `argv[0]` der Name des aufgerufenen Programmes. In den weiteren null terminierten Zeichenketten `argv[1]` bis `argv[argc-1]` sind dann die beim Aufruf übergebenen Argumente zu finden.

B.4 Lektion 6 — Konstruktoren und Destruktoren

Aufgabe 6.1

a Um den parameterlosen Konstruktor der Klasse `integer` aufzurufen, muß die Anweisung `integer xyz;` verwendet werden.
Hingegen deklariert die Anweisung `integer xyz();` eine parameterlose Funktion mit einem Rückgabewert vom Typ `integer`.

b Der erste Aufruf erzeugt dynamisch **eine** Instanz vom Typ `int`. Hierbei wird der Kopierkonstruktor aufgerufen und `*x` mit `5` initialisiert.
Bei der zweiten Anweisung wird dynamisch ein Feld mit **fünf** Instanzen vom Typ `int` angelegt. Für jedes Objekt wird dabei der parameterlose Standardkonstruktor aufgerufen.

c

```
person::person(const string& n,
 const string& a) {
  name = n;
  address = a;
}
```

Bei der Initialisierung der klassenspezifischen Variablen `name` und `address` wird zuerst für diese der parameterlose Standardkonstruktor aufgerufen (geschieht implizit) und dann der Zuweisungsoperator auf die übergebenen Argumente des Konstruktors angewendet.

```
person::person(const string& n,
 const string& a): name(n),
 address(a) {}
```

Hierbei handelt es sich um einen Konstruktor mit Initialisierungsliste, d.h. es wird nicht wie oben der parameterlose Konstruktor für `name` und `address` aufgerufen sondern der entsprechende, der diese klassenspezifischen Variablen gleichzeitg initialisert.

```
person& person::operator=(const
 person& p) {
  if (this!=&p) {
   name = p.name;
   address = p.address;
  }
  return *this;
}
```

Aufgabe 6.2

a Kopiervorgänge bei der Rückgabe eines Funktionswertes zu vermeiden, ist i.allg. nur dadurch möglich, daß innerhalb der Funktion dynamisch Speicher allokiert wird. In der `return` Anweisung kann dann ein Zeiger auf dieses Objekt zurückgegeben werden.

```
string* do_something(const
 string& v) {
  // tue etwas

 string* x=new
       string("Professor");

 return x;
}
```

Aufgabe 6.3

a Die Funktion `func` erwartet als Argument ein `string` Objekt. Hierzu wird **implizit** der `string::string(const char*)` Konstruktor aufgerufen.

b Die Funktion `funcky` liefert ein Objekt vom Typ `string` zurück. Um aus ¨`Torvalds`¨ ein `string` Objekt zu erzeugen, wird der `string::string(const char*)` Konstruktor aufgerufen. Anschließend wird noch der Kopierkonstruktor `string::string(const string&)` aufgerufen, da der Rückgabewert kopiert wird.

c Mit diesem weiteren Konstruktor ist es möglich, ein `string` Objekt zu instanziieren und es mit einem `int` Objekt zu initialisieren. Hierzu wird `i` als Zeichenkette interpretiert (`sprintf`).
Die erste Anweisung ist äquivalent zu `string s1(7);` , d.h. es wird keine Zuweisung vorgenommen sondern eine Initialisierung mit dem `string::string(int)` Konstruktor.
Bei der zweiten Anweisung wird in der Tat eine Zuweisung ausgeführt. Zuerst wird für `s4` der parameterlose Standardkonstruktor aufgerufen. Anschließend wird dann eine temporäre Instanz erzeugt, d.h. mit dem `string::string(int)` ein solches Objekt instanziiert. Abschließend wird dann `s4` das temporäre Objekt zugewiesen.

Aufgabe 6.4

c

```
#include <iostream.h>

struct X {
 X() {
  cout << "Programmieren in C++"
         " an der" << endl; }

 ~X() {
  cout << "macht viel Spass."
       << endl; }
} Y;
  // Y ist globales Objekt !

int main() {
 cout << "Uni Hildesheim"
      << endl;
}
```

B.5 Lektion 7 — Vererbung

Aufgabe 7.1

a Zuerst werden die Konstruktoren der Basisklassen in der Reihenfolge entsprechend der **Klassendeklaration**[3] aufgerufen und initialisiert. Anschließend werden die klassenspezifischen Variablen in Deklarationsreihenfolge initialisiert. Hieran folgt die Ausführung der Anweisungen im Konstruktorrumpf. ISO 12.6.2
Der Aufruf der Destruktoren für klassenspezifische Variablen und die der Basisklassen erfolgt in umgekehrter Reihenfolge der Konstruktion. Zuerst werden allerdings die Anweisungen im Rumpf des Destruktors ausgeführt. ISO 12.4

c

[3] Die Reihenfolge der im Konstruktor aufgeführten Initialisierungsliste ist unerheblich !

```
class W;
class X;
class Y;

class Z : public X, protected Y,
          private W {

 /* Klassendeklaration */

}
```

ISO 11.2 Durch die Angabe von `public`, `protected` oder `private` in der Klassendeklaration wird der Zugriff auf Methoden und klassenspezifische Variablen der Basisklasse geregelt.

`public` gibt an, daß auf die `public` Methoden und klassenspezifischen Variablen der Basisklasse im `public` Bereich der abgeleiteten Klasse zugegriffen werden kann. Entsprechendes gilt für die `protected` Methoden und Attribute der Basisklassen.

`protected` gibt an, daß auf die `public` und `protected` Methoden und klassenspezifischen Variablen der Basisklasse im `protected` Bereich der abgeleiteten Klasse zugegriffen werden kann.

`private` gibt an, daß auf die `public` und `protected` Methoden und klassenspezifischen Variablen der Basisklasse nur im `private` Bereich der abgeleiteten Klasse zugegriffen werden kann.

d

```
class counter : public gauge {
 public:

  counter(int,int,int);

  void set(int);
};

inline counter::counter(int val,
 int a, int b) : gauge(val,a,b)
{}

inline void counter::set(int
 val) {
  value=(val>max ? min : val);
}
```

B.6 Lektion 8 — Virtuelle Methoden und abstrakte Basisklassen

Aufgabe 8.1

a `integer` ist eine `public` Basisklasse von `gauge`. Um zu vermeiden, daß auf `public` Methoden von `integer` zugegriffen werden kann, muß `integer` eine `protected` oder `private` Basisklasse sein.

Literaturverzeichnis

[Boo91] Grady Booch. *Object Oriented Design with Applications*. The Benjamin/Cummings Publishing Company, Redwood City, California, 1991.

[BT89] Falko Bause and Wolfgang Tölle. *Einführung in die Programmiersprache C++*. Vieweg, Braunschweig, 1989.

[Cop92] James O. Coplien. *Advanced C++ — Programming Styles and Idioms*. Addison–Wesley, Reading, Massachusetts, 1992.

[CR91] Debra Cameron and Bill Rosenblatt. *Learning GNU Emacs*. O'Reilly & Associates, Inc., Sebastopol, CA, 1991.

[CY91a] Peter Coad and Edward Yourdon. *Object-Oriented Analysis*. Yourdon Press/Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[CY91b] Peter Coad and Edward Yourdon. *Object-Oriented Design*. Yourdon Press/Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[Dav94] Stephen R. Davis. Armor cladding C++ classes. *C++ Report*, 6(8):36–41, Oct 1994.

[Dew89] Alexander K. Dewdney. Die elektronische Beziehungskiste, Computer–Kurzweil III. *Spektrum der Wissenschaft (Sonderheft)*, 8, 1989.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison–Wesley, Reading, Massachusetts, 1990.

[Fel79] Stuart I. Feldman. Make – a program for maintaining computer programs. *Software – Practice and Experience*, 9:255–265, 1979.

[Gul88] Jürgen Gulbins. *UNIX*. Springer–Verlag, Berlin Heidelberg, 1988.

[HHMK94] Sebastian Hetze, Dirk Hohndel, Martin Müller, and Olaf Kirch. *Linux Anwender Handbuch*. LunetIX Softfair, Berlin, 1994.

[Hil91] Universiät Hildesheim. vi–Handbuch. 1991.

[HS91] Samuel P. Harbison and Guy L. Jr. Steel. *C — Ein Referenzhandbuch*. Wolfram's Fachverlag, 1991.

[ISO95] ISO WG21/ ANSI X3J16, `ark@research.att.com`. *Working Paper for Draft Proposed International Standard for Information Systems– Programming Language C++*, April 1995.

[Koe94] Andrew Koenig. Hints for keeping bugs away. *C++ Report*, pages 37–39, 1994.

[Koe95] Andrew Koenig. Wrapping up the standard. *Journal of Object Oriented Programming*, pages 60–63, Jun 1995.

[KR83] Brain W. Kernighan and Dennis M. Ritchie. *Programmieren in C*. Carl Hanser Verlag, München, 1983.

[Lip91] Stanley B. Lippman. *C++ Primer*. Addison–Wesley, Reading, Massachusetts, second edition, 1991.

[LS94] Horst Langendörfer and Bettina Schnor. *Verteilte Systeme*. Carl Hanser Verlag, München, 1994.

[Mey90] Bertrand Meyer. *Objektorientierte Softwareentwicklung*. Carl Hanser Verlag und Prentice–Hall International, Wien (Hanser), London (Prentice–Hall), 1990.

[Mün95] Heiko Münkel. Xemacs 19.13 – mehr als nur ein Editor. *iX*, 11:94–105, 1995.

[OT91] Andrew Oram and Steve Talbott. *Managing Projects with make*. O'Reilly & Associates, Sebastopol, California, 1991.

[RBP+93] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Objektorientiertes Modellieren und Entwerfen*. Hanser und Prentice–Hall, München, Wien: Hanser; London: Prentice–Hall, 1993.

[SL94a] A. Spillner and P. Liggesmeyer. Software–Qualitätssicherung in der Praxis. *Informatik Spektrum*, 17:368–372, Dez 1994.

[SL94b] Alexander Stepanov and Meng Lee. The Standard Template Library. Technical report, Hewlett–Packard Laboratories, Palo Alto, California, 1994. `ftp://butler.hpl.hp.com`.

[Sto95] Christoph Stockmayer. Helferlein. *iX*, 6:180–182, 1995.

[Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison–Wesley, Reading, Massachusetts, second edition, 1991.

[Str94a] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison–Wesley, Reading, Massachusetts, 1994.

[Str94b] Bjarne Stroustrup. Making a vector fit for a standard. *C++ Report*, pages 30–34, Oct 1994.

[Vil94] Michael J. Vilot. An introduction to the Standard Template Library. *C++ Report*, pages 22–29, 35, Oct 1994.

[Wag95] Michael P. Wagner. Katze aus dem Sack. *iX*, pages 152–158, Jun 1995.

Index